NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

OPTIMAL THREE-DIMENSIONAL PATH PLANNING
USING VISIBILITY CONSTRAINTS

by

David Hugh Lewis

December 1988

Thesis Advisor: Neil C. Rowe

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 33	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Optimal Three-Dimensional Path Planning Using Visibility Constraints

12 PERSONAL AUTHOR(S)
Lewis, David H. (the author)

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	December 1988	229

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial-Intelligence, Spatial reasoning, paths, planning
Snell's Law, (ICR)

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

~~We~~ presents an algorithm for finding optimal three-dimensional paths above polyhedral models of terrain. ~~Airspace~~ is modeled as irregularly-shaped regions of homogeneous probability-of-detection, with respect to one or more fixed observers. ~~We~~ plans paths by first finding an optimal set of contiguous visibility regions, then an optimal piecewise-linear flight path through this envelope, using Snell's Law to find locally-optimal maneuver points. The performance of our region-finding algorithm favorably compares with an alternate approach using regular cubic regions.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL
Neil C. Rowe	(408) 646-2462	52Rp

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited

Optimal Three-Dimensional Path Planning
Using Visibility Constraints

by

David Hugh Lewis
Lieutenant, United States Navy
B. S., University of Nebraska Lincoln, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

Author: _____
David Hugh Lewis

Approved by: _____
Neil C. Rowe, Thesis Advisor

_____
Yuh-Jeng Lee, Second Reader

_____
Robert B. McGhee, Chairman
Department of Computer Science

_____
Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

We present an algorithm for finding optimal three-dimensional paths above polyhedral models of terrain. Airspace is modeled as irregularly-shaped regions of homogeneous probability-of-detection, with respect to one or more fixed observers. We plan paths by first finding an optimal set of contiguous visibility regions, then an optimal piecewise-linear flight path through this envelope, using Snell's Law to find locally-optimal maneuver points. The performance of our region-finding algorithm favorably compares with an alternate approach using regular cubic regions.

TABLE OF CONTENTS

I. INTRODUCTION

Path planning is an important and growing area of research in artificial intelligence. Recent research has produced an extensive and sophisticated body of knowledge. Much of this work has been restricted to the (essentially) two-dimensional path planning problem for a ground vehicle moving on ground terrain. However, many interesting and potentially useful path-planning problems involve true three-dimensional paths. Examples include aircraft flight-paths, submersible paths underwater, overhead-crane paths within large factories, tunneling plans for underground shaft mines, and routes for microwave communications links. Many of these areas have potential for application within the U. S. Navy.

Unfortunately, the three-dimensional path-planning problem is not a simple extension of the two-dimensional path-planning problem. Many physical restrictions exist which constrain the three-dimensional problem in ways which are fundamentally different from the constraints found in two-dimensional problems (it is generally not possible to stop in mid-air while flying, for instance). Certain abstractions may be made to simplify the problem. If one thinks of the two-dimensional path planning problem as finding a path through a flat, two-dimensional graph of a given precision (or granularity), then a three-dimensional path planning problem can be thought of as a search through a

three-dimensional graph, again of a given granularity. This graph is the **search space**.

If one considers a cube of air 1000 meters on a side as a search space, dividing it into 10 meter cubes will produce a search graph containing one million nodes. Using 25-meter cubes, 64 thousand nodes will be needed to define the search graph. Using 50-meter cubes, eight thousand nodes will be needed This is illustrated in Figure 1.

To handle this over-abundance of search space, certain assumptions can be made concerning the type of path desired and the nature of the search graph. One method is to reduce the volume of the search space by aggressive pruning (e.g., making assumptions) as the search progresses. This thesis proposes an alternate method for reducing the size of the search space, where, by coalescing nodes sharing some common characteristic, the size of the search space is reduced before the search starts. Other characteristics (such as travel cost, altitude, cloudiness, humidity, for example) of the original nodes must then be compensated for while performing the actual search. To illustrate this approach, a subclass of the generalized three-dimensional routing problem was selected for analysis. Only two factors were used in optimization while path planning through the three-dimensional search graph: visibility (probability-of-detection), and cost (or energy expended). Visibility will be used as the **grouping characteristic**, and cost will be used

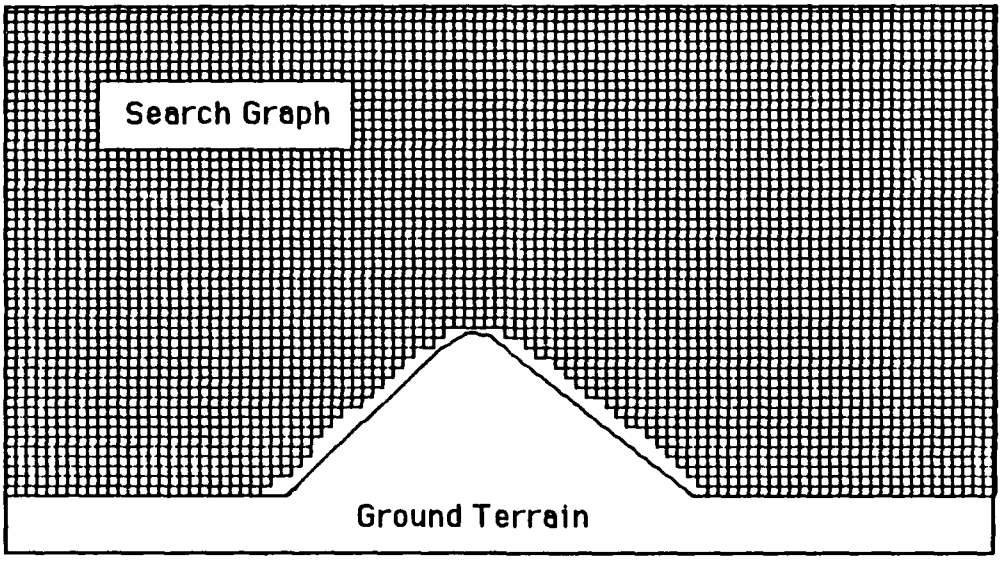


Figure 1. Search Graph for Simple 3-D Search in Cross-Section

as the **compensating characteristic** which must be compensated for as the search progresses. All path planning will be conducted over idealized, polyhedral approximations of terrain.

This thesis will thus consider three-dimensional routing problems which plan routes between two given points in space over terrain modeled as a polyhedron, optimizing the route for low probability-of-detection and cost, given a finite number of immobile observers to generate the visibility criteria. This subclass of problems includes such "real world" examples as route planning for cruise missiles from launch point to final approach on target; routing of manned attack aircraft to (and from) a mission objective; the routing of a submersible through a field of active sonar devices; and detecting "holes" in the area air defense of a high value target. Optimizing for high probability-of-detection instead of low probability-of-detection has application to another class of problems, such as routing of commercial passenger aircraft between radar sites, maintaining a Remotely Piloted Vehicle within line-of-sight (LOS) communication of one or more controllers, or ensuring that ground-based autonomous vehicles remain within LOS communication of one or more communication sites for as long as possible.

Given a piece of terrain, and an observer, as in Figure 2, the airspace can be divided into two regions: airspace

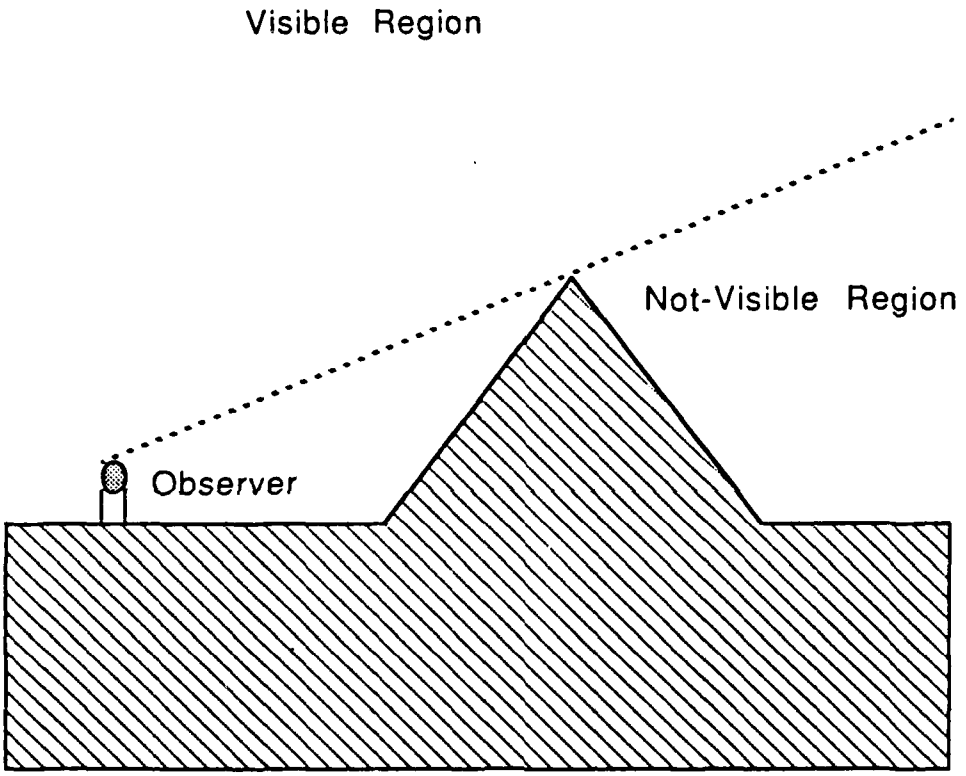
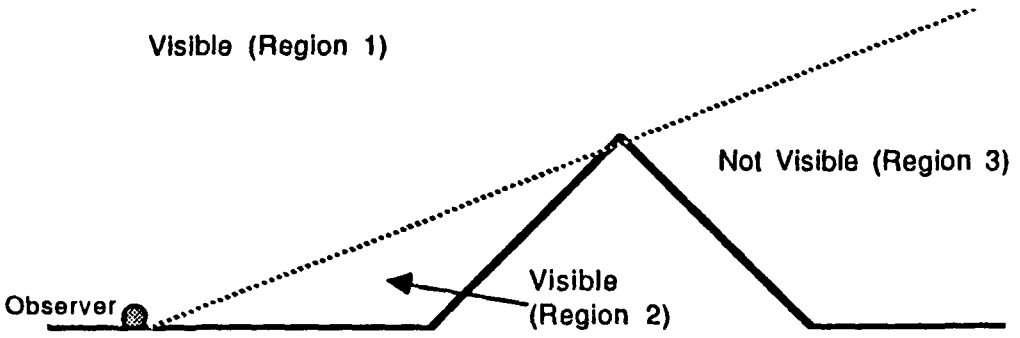


Figure 2. The Concept of Visibility Regions

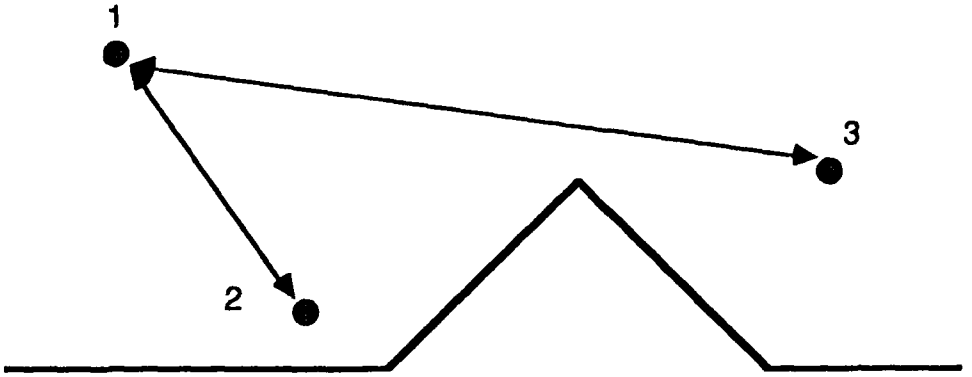
that is visible to the observer, and airspace which is not. These regions can be determined by a simple construction involving the observer and the terrain. All of the air within a given **visibility region** will be considered as a single entity when discussing visibility of that region; any one point within the region is no more or less visible than any other point. Using this argument, then, the airspace above the observer in Figure 3 can be reduced to a very simple graph when evaluating visibility. In terms of Figure 3b and 3c, being "not visible" means being at graph node 3; being "visible" means being at graph node 1 or 2. This abstraction of the visibility of space will be a powerful tool in the three-dimensional path planning which follows. Figure 4 illustrates a more complex example.

The second aspect of the class of path planning considered here involves minimizing the energy expended to get between two points. Energy costs are a function of location and velocity. These costs depend upon the scale of the terrain, and the type of device which is expected to follow the final path. To simplify matters, a more "generic" approximation based on distance travelled was developed, using linear corrections to account for energy lost or gained while turning, diving and climbing.

Path analysis for the problems discussed above involves combining visibility information obtained from a visibility model with energy costs in an appropriate manner. An optimal



Visibility Regions
(a)



Search Graph with Spatial Relationships
(b)

Abstract Search Graph
(c)

Figure 3. Construction of a Search Graph

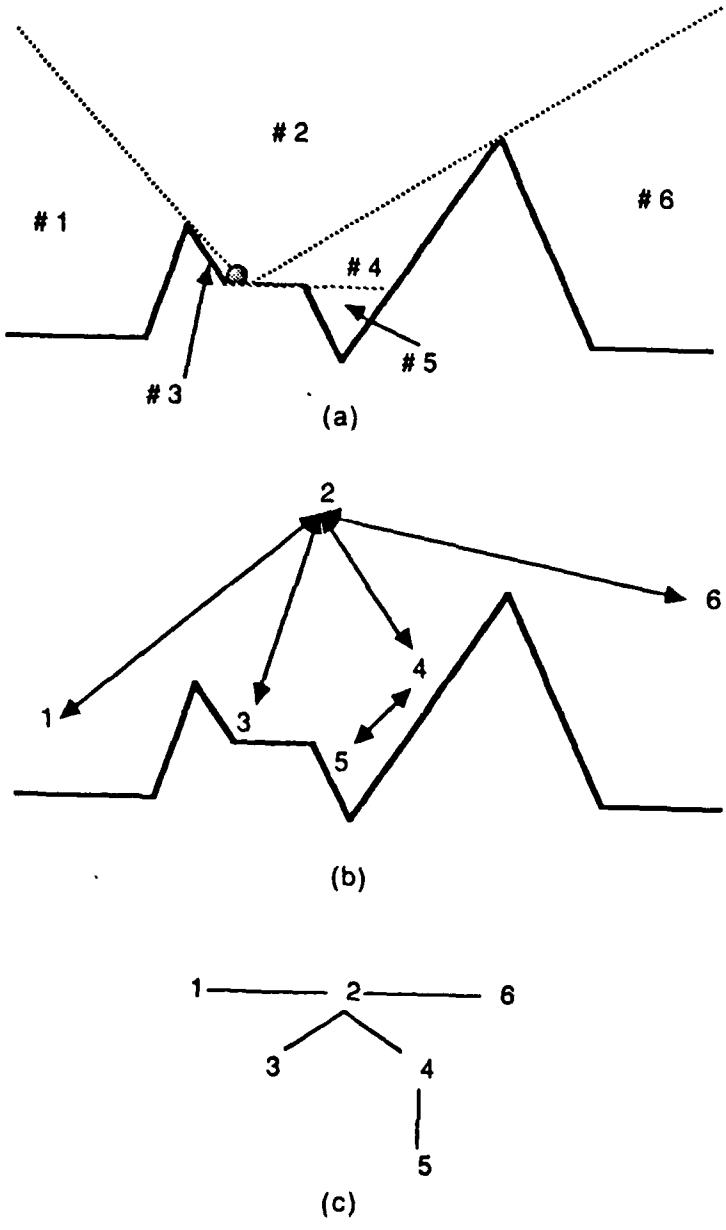


Figure 4. A More Complex Search-Graph Construction

path, or "best path," minimizes the weighted average of the probability-of-detection and energy that must be expended. In general, this optimal path will not be the same as either the minimum energy path (ignoring visibility), or the minimum probability-of-detection path (ignoring cost).

An artificial-intelligence search method called A* search can perform the visibility-region sequence-planning within the context of the visibility and cost modelling techniques discussed above. However, the path-planning problem is not then completely solved, since A* search will produce a contiguous series of regions (volumes) with common visibility characteristics shown in Figure 5, called a **volume path**. An optimal **piecewise linear path** must still be found within the confines of the volume path produced by the A* search, such that the path is linear within each volume. Determination of the optimal piecewise-linear flight path through the volume path reduces to an optimization problem in three-dimensions. Fermat's Principle from optics can be used to solve this optimization problem.

The remaining sections of this thesis describe an implementation of the three-dimensional path planning approach discussed above. Chapter II introduces all relevant background topics, including solid geometry, A* search and Fermat's Principle. Chapter III describes the visibility-grouping and path-planning algorithms in detail, while Chapter IV describes the details of their algorithms.

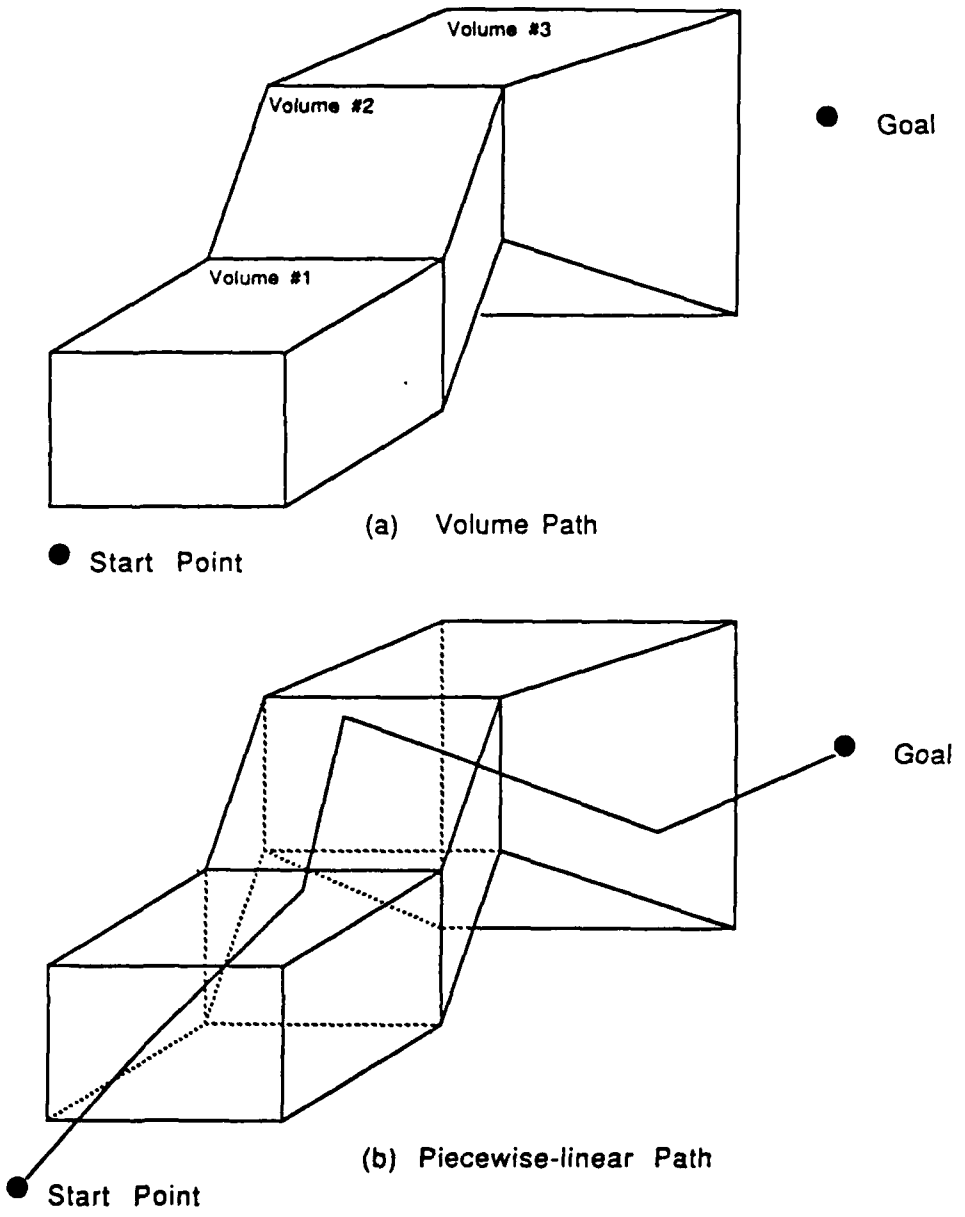


Figure 5. Types of Paths

Results of the implementation will be presented in Chapter V, which shows both the visibility models and some representative flight paths over a variety of terrain features. Finally, Chapter VI discusses the strengths and weaknesses found in the algorithms and their implementations.

II. BACKGROUND

A significant amount of previous work has been done in path-planning, although very little of it has involved three-dimensional path-planning [Ref. 1]. A family of generalized search algorithms has been developed [Ref 2] which can be applied to almost any sort of search problem when the search nodes are discrete and ennumerable. These generalized search methods have been successfully applied to various two-dimensional path-planning problems [Ref. 3], [Ref. 4] and [Ref. 5], and two of these generalized search methods will be used in the three-dimensional path-planning problem presented in this thesis. Only a small body of published literature discusses three-dimensional path-planning, and that addresses rather abstract versions of the problem (for example, [Ref. 6] discusses path-planning along the surface of a solid convex volume).

A. RELATED PROBLEMS AND SOLUTIONS

1. General Predefined-Node Search Algorithms

[Ref. 2] contains a description of all generalized search algorithms. Table 9-5 from that reference is reproduced as Table 1, illustrating most of the currently used generalized search algorithms, and some of their features. All of these search functions seek to find some path (often an optimal path, but not always) between a given

TABLE 1. SEARCH METHODS

Name of search strategy	*Uses agenda?*	*Uses evaluation function?*	*Uses cost function?*	*Next state whose successors are found*
Depth-first search	no	no	no	A successor of the last state, else a successor of a predecessor
Breadth-first search	yes	no	no	The state on the agenda the longest
Hill-climbing (optimization)	no	yes	no	The lowest-evaluation successor of the last state
Best-first search	yes	yes	no	The state on the agenda of lowest evaluation value
Branch-and-bound	yes	no	yes	The state on the agenda of lowest total cost
A* search	yes	yes	yes	The state on the agenda of lowest sum of evaluation value and total cost

* Reprinted from Rowe, Neil C., Artificial Intelligence Through Prolog, Prentice-Hall, 1988

start point to a given goal point, through a finite search space. The search space may be a representation of physical objects, such as a graph representing terrain, or it may represent more abstract concepts, such as the set of all possible steps in fixing a car [Ref. 2]. Most searches that find optimal paths (A*, Best-first, Branch-and-bound), use some sort of metric to judge the distance to either the start point, goal point or both. These metrics are then used to decide which paths through the search space are optimal, or lowest cost. These metrics are called cost functions and evaluation functions.

A cost function is a function which produces a number representing the effort (or cost, or energy expended) to get to a given point in the search space from the start point. An evaluation function is similar to a cost function, except that it produces a number which indicates the expected cost to get to the goal from a given location in the search space. A* search uses the sum of the cost and evaluation functions to decide which paths are optimal (e.g., lowest total cost) as the search progresses through the search space towards the goal. Because of this, A* search is generally regarded as the best of the search methods, even though it is the most complicated [Ref. 2]. Other search methods can be used if optimality can be sacrificed, or if the particular problem at hand can be best solved using a less sophisticated search algorithm.

The three-dimensional path planning program uses A* search for path-planning through the visibility volumes, and the best-first search in one application where the cost function has no meaning.

2. Path Planning in Two-Dimensions

Current work in two-dimensional path planning can be divided into three major approaches; path-oriented approaches, spatial-reasoning based algorithms and blind search methods [Ref. 7]. The path-oriented approach uses a search space of all possible paths (developed by some other search method) from the start to the goal, and then selects the best path from this list using concepts from decision theory [Ref. 2]. The spatial reasoning approach reasons about how paths are affected by obstacles. An example of this is the visibility graph. Visibility graphs, or V-graphs, have a search space composed of lines of mutual visibility between edges and vertices of obstacles. Path planning is accomplished along these lines, from the start point to the goal. Blind searches generally use a form of wavefront propagation search, which subdivides the search area into small squares, and then finds paths by looking, one square at a time, in all directions, until the goal is reached. The search graph in this search consists of the squares as nodes, and the connections between squares as the edges of the graph. Most of these methods can be adapted to

the three-dimensional path-planning problem in one form or another.

3. Path Planning in Three-Dimensions

The current approaches to the three-dimensional path-planning problem center on a three-dimensional version of the two-dimensional visibility graphs or involve attempts to decompose the three-dimensional problem to a two-dimensional problem [Ref. 6]. This thesis is unique in approaching the problem by reducing the size of the search space by grouping possible graph nodes, and then applying a simple search to find an optimal path.

4. Fermat's Principle

Fermat's Principle, first proposed by Pierre de Fermat in 1657, states that light rays travelling through an optical medium will always take the path which can be traveled in the least amount of time [Ref. 8]. As light passes through several layers of different optical mediums, then, it will travel along a path with the shortest optical path length. Fermat's Principle leads to Snell's Law, which states that

$$N1*\sin\beta 1 = N2*\sin\beta 2 \qquad (2\text{-}1)$$

where

N1 is the index of refraction of material #1,

ß1 is the angle of incidence, measured from the normal,

N2 is the index of refraction of material #2, and

ß2 is the angle of refraction, measured in a counter-clockwise sense from the normal.

Figure 6 illustrates this effect. Fermat's Principle provides a useful method of minimizing the total transit time of a path between start and goal points. Assuming that the flight path is flown at a constant speed, this minimized transit time can be multiplied by the speed (a constant) to produce a minimized distance:

$$\text{Distance} = \text{Time} * \text{Speed} \qquad (2\text{-}2)$$

As a result, all Snell's Law optimizations will be discussed as optimization of distance, even though time is actually the variable optimized. Note that this is only true if the speed is a constant, and if no other costs are accounted for at the turn-point. If turn costs were to be included in the Snell's Law optimization, then Equation 2-2 will not hold.

The index of refraction used in Snell's Law can be constrained to be the probability-of-detection of a particular visibility volume, so that a path through a series of visibility volumes can be considered analogous to a light ray passing through a series of optical mediums. Minimizing the deviation from Snell's Law at each transition layer between volumes will guarantee minimization of the total path

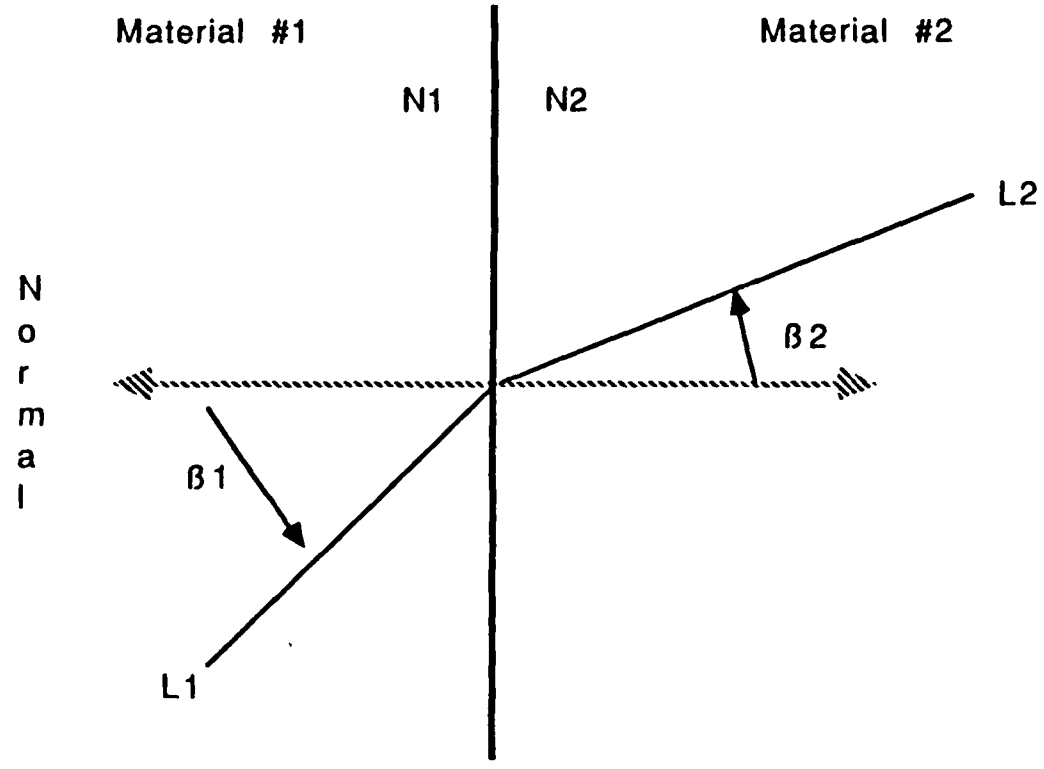


Figure 6. Snell's Law

length because minimization of Equation 2-1 is convex [Ref. 4].

5. Visibility Models

A significant amount of prior research has been done in the area of visibility and visibility algorithms. The three-dimensional path-planning program requires an all-around visibility model. Fortunately, some aspects of standard graphics approaches to visibility can be modified for use for this model. [Ref. 10] refers to finding shadow regions of polyhedral solids by projecting planes from the observer to the edges of a shadowed object, creating a shadow polygon behind the shadowed object.

B. MATHEMATICS BACKGROUND

1. Planes

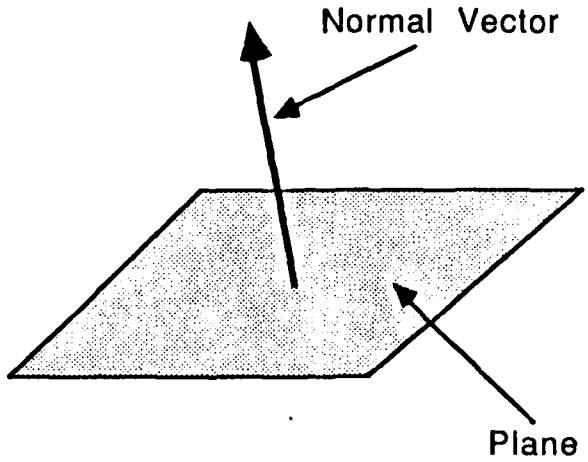
A quick overview of some basic three-dimensional mathematics is in order at this point. A plane in space is represented by an equation of the form

$$Ax+By+Cx=Ao \qquad (2\text{-}3)$$

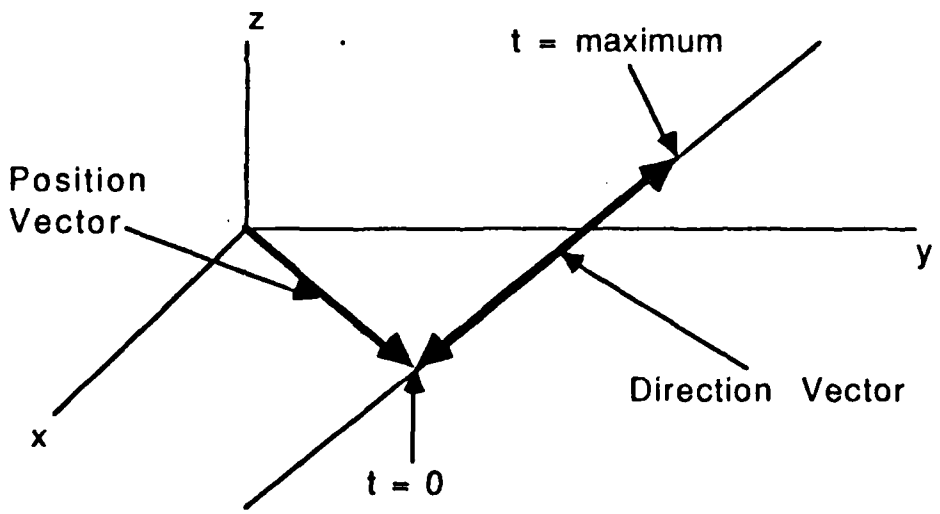
where A, B and C are the coefficients of the vector N normal to the surface of the plane.

$$N=Ai+Bj+Ck \qquad (2\text{-}4)$$

and Ao is found by substituting the coordinates of any point in the plane into the left hand side of Equation 2-3 [Ref. 11]. Figure 7a illustrates this arrangement.



Plane Representation
(a)



Line Segment Representation
(b)

Figure 7. Lines and Planes

2. Lines

A line in space has a more complicated representation. Any line in space is completely defined by two vectors: a position vector leading from the origin to any point on the line, and a direction vector pointing along the direction of the line. The coordinates of any point along the line can be determined by changing the coefficient (or parameter) t in the line equation

$$P(x,y,z)=A+B+C+t*(D+E+F) \qquad (2\text{-}5)$$

where A, B and C are the coefficients of the position vector, and D, E and F are the coefficients of the direction vector

$$\text{Position Vector}=Ai+Bj+Ck \qquad (2\text{-}6)$$

$$\text{Direction Vector}=Di+Ej+Fk. \qquad (2\text{-}7)$$

This is called the parametric form of a line in space [Ref. 11], and is illustrated in Figure 7b.

The parametric form of a line is very useful since any single point along the line may be specified by a single parameter, rather than two or three different variables. A line segment can thus be easily defined by means of two parameters, one each for the endpoint of the line segment. If the parameter at one endpoint is arbitrarily set to zero, then the entire line segment can be represented by a single parameter and some constants.

3. Volumes

Figure 8 illustrates the structure of solid polyhedra, or volumes, used in this thesis. The relationships between volumes, points, edges, facets and planes can be best illustrated by the hierarchical graph of Figure 9. These relationships form the foundation for the subsequent development of the data types to be used to represent these objects during the implementation of the path planning program. These data structures will be presented in detail in Chapter IV.

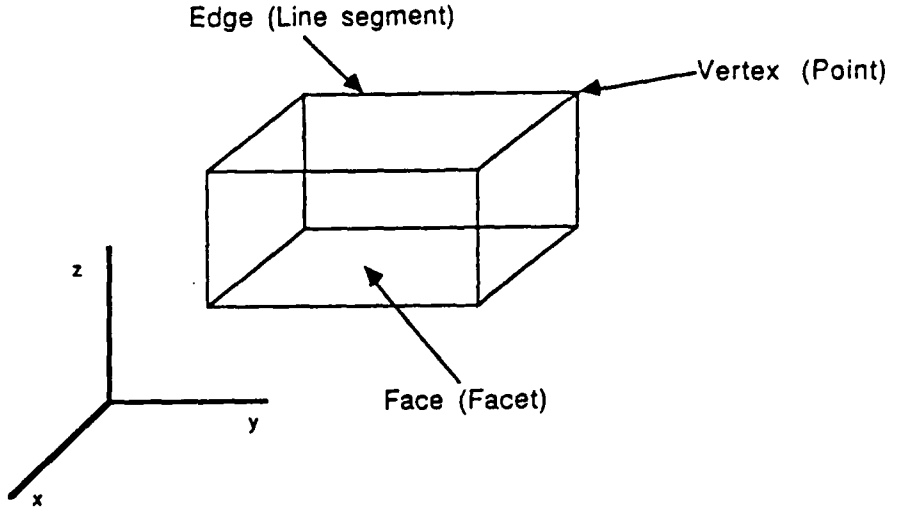


Figure 8. Volume Structure

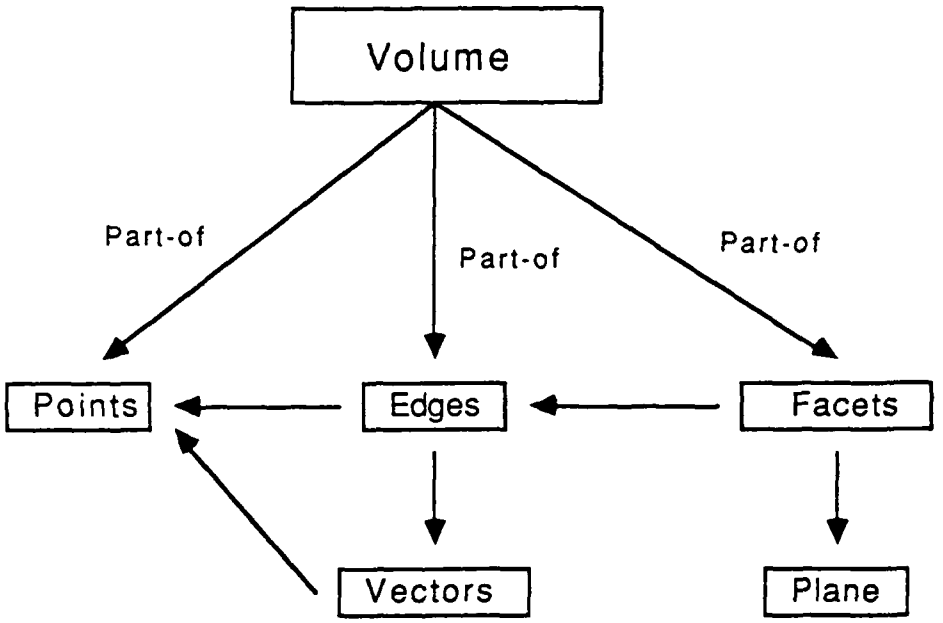


Figure 9. Relationships Among Solid-Geometry Data Structures

III. VISIBILITY AND PATH-PLANNING ALGORITHMS

This chapter will describe the algorithms used for three-dimensional path planning. Most of this chapter will discuss visibility determination and path planning. Later sections will present algorithms in solid geometry.

As discussed in Chapter I, a fundamental concept in this work is the application of a grouping, or coalescing criteria to three-dimensional volumes in the three-dimensional path planning problem, using one or more common characteristics within large regions of airspace to simplify the search problem. The common characteristic used here will be visibility.

The three-dimensional path-planning algorithm breaks naturally into two halves: **static-visibility determination** and path planning. The static-visibility determination phase uses the grouping characteristic (visibility) to find visibility volumes, and then builds a search graph. Path planning performs the search through the search graph, which generates the volume path defined in Chapter I. The path-planning algorithm must then find the final optimal piecewise-linear path through the volume path.

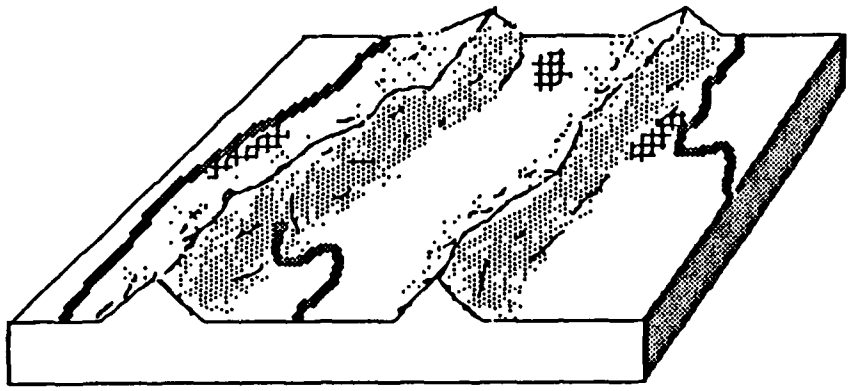
A. VISIBILITY-REGION CONSTRUCTION

1. Terrain Model

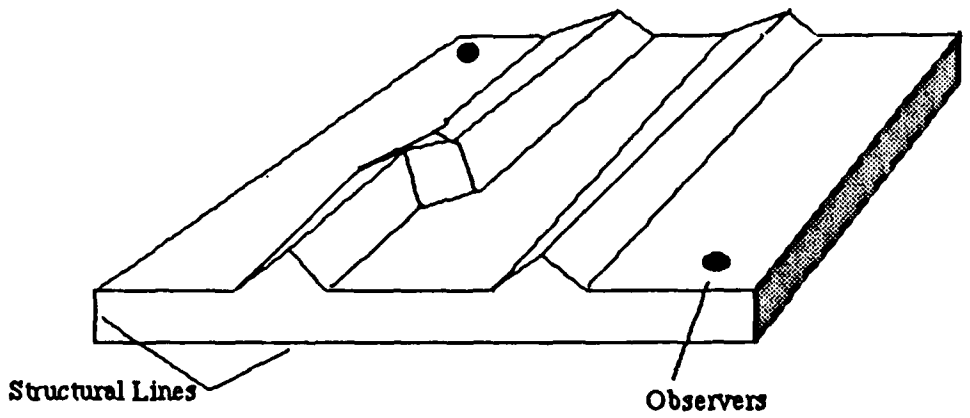
A polyhedral model of terrain will be used for path planning to simplify the mathematics of the visibility algorithms. The terrain model used is described in [Ref. 9]. This type of terrain model produces large planar patches which are as large as possible given the terrain, not the simple triangular terrain patches found in more common polyhedral models. Figure 10a shows some sample real terrain, and Figure 10b shows the more general polyhedral model used. Figure 10b also shows the structural lines used to create a base for the terrain model. These vertical and horizontal edges simply outline the sides and bottom of the terrain to create a ground volume, which can be manipulated in the same way that any air volume can be manipulated. Creation of these lines is discussed in the following chapter.

2. Static Visibility-Determination Algorithm

The location and extent of the visibility regions is completely determined by the geometry of the terrain and the location of observers relative to that terrain. Regions of visibility (regions of non-zero probability-of-detection) are regions of points which have an unobstructed line-of-sight (LOS) from an observer. Figure 11 shows cross-sections of two different examples of such visibility regions.

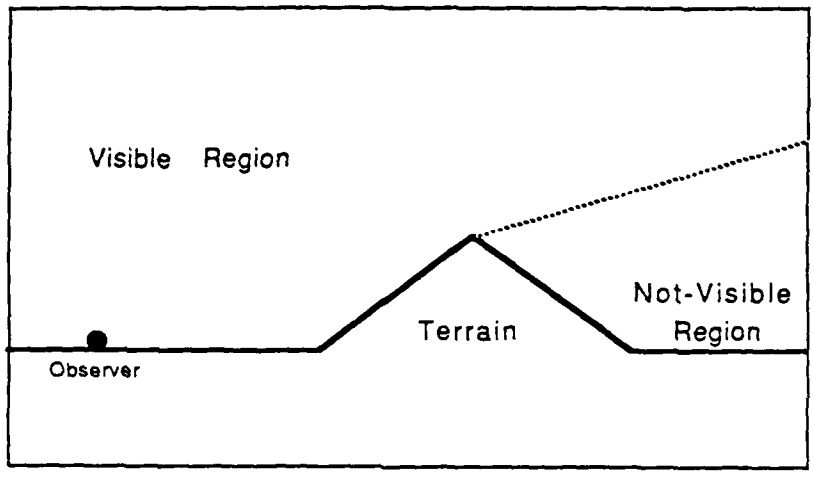


(a)

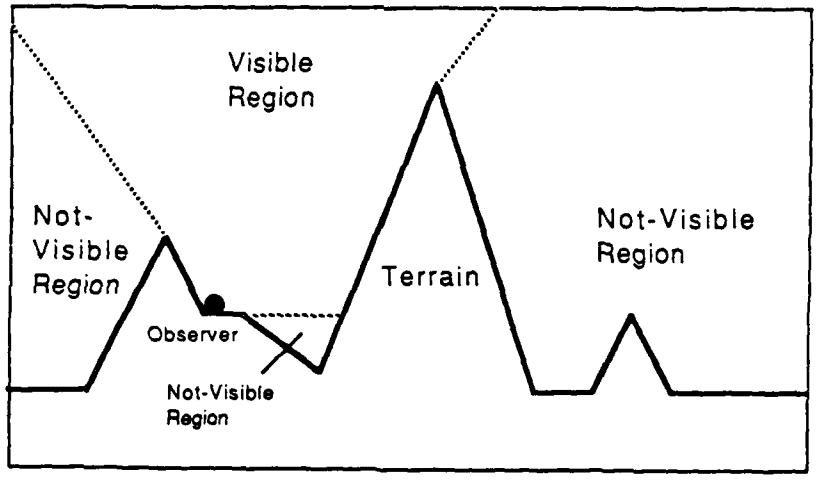


(b)

Figure 10. Terrain Modeling



(a)



(b)

Figure 11. Visibility Regions

a. Determination of Visibility Regions

Non-visible regions (with respect to a single observer) are simply regions shadowed from the observer by some terrain feature. Many shadowing algorithms are available, as discussed in Chapter Two, but most are oriented towards views of terrain from particular vantage points, and generally do not involve construction of shadow volumes. some shadowing algorithms [Ref. 10] do provide useful methods which can be extended to the full shadowing problem with great success. These methods involve constructions between an observer and significant features of the surrounding terrain. Note in Figure 11 that the limits of the visibility regions are determined by a linear construction between the observer and the points of peaks. In three dimensions (using polyhedral approximations to real terrain), these peaks would be ridge lines, and the linear construction would be a plane (see Figure 12) called the **limiting plane of visibility**).

This limiting plane of visibility is easily found, since a plane can be defined by a point (the observer) and a line (the ridge line). As a result, the visibility regions may be built using the following algorithm:

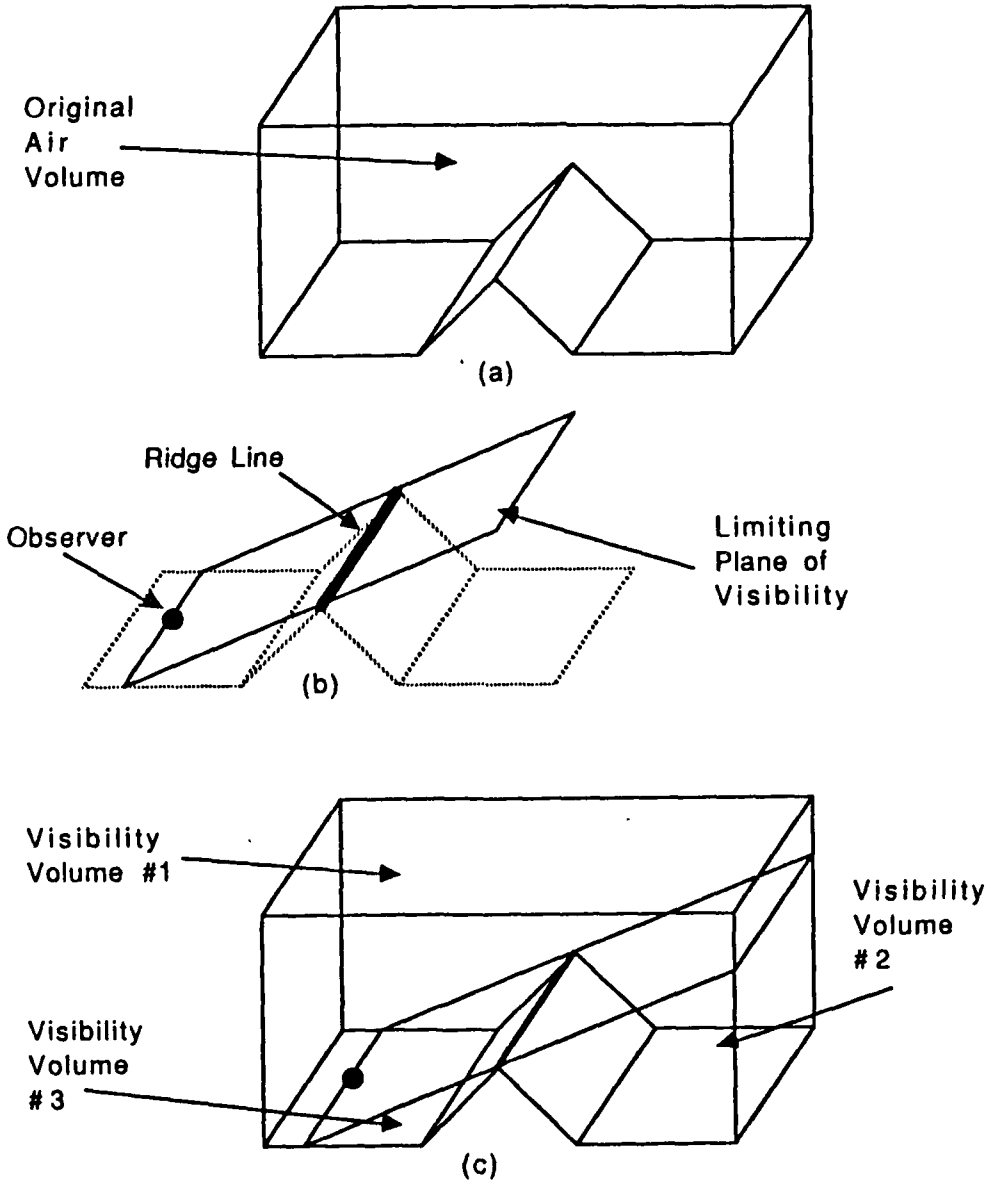


Figure 12. Construction of Visibility Volumes

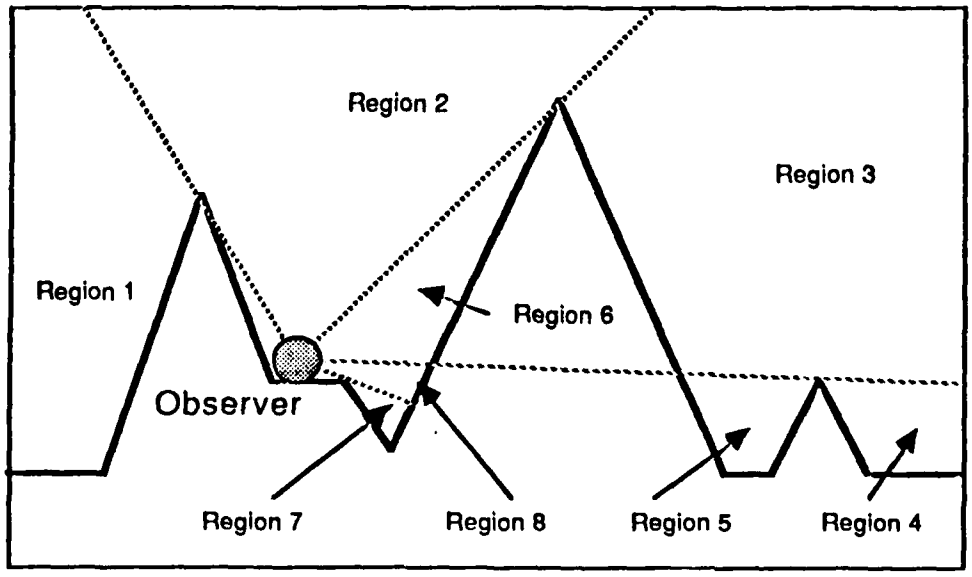
```
Initial Condition:  A large volume of air above some terrain
    LOOP for Observer IN (all observers)
        LOOP for Ridge-line IN (all ridge-lines)
            CONSTRUCT Limiting Plane of Visibility using
                      Observer and Ridge-line
            DIVIDE all volumes at the Limiting Plane of
                   visibility
        END
    END
```

Algorithm 3-1. Construction of Visibility Volumes

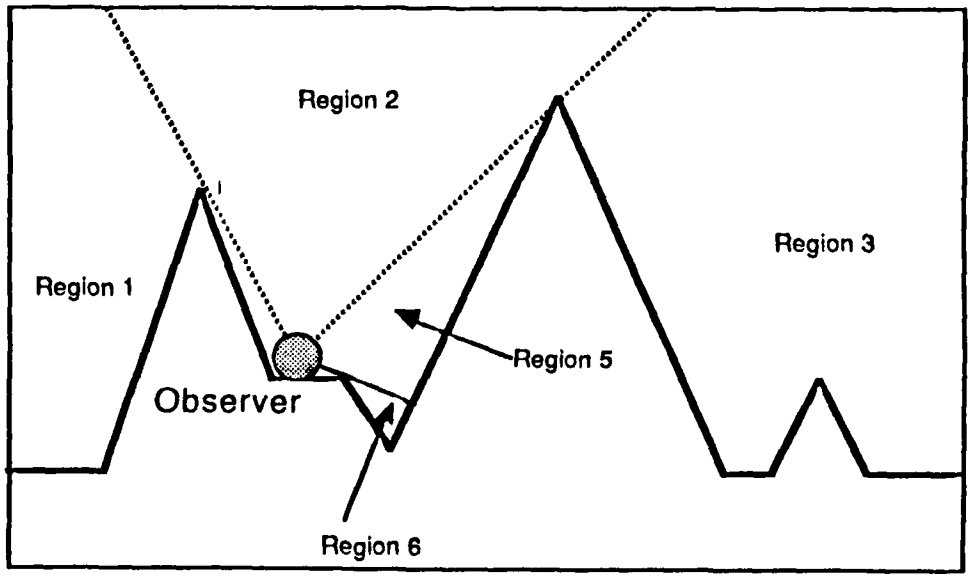
Results of this algorithm are illustrated in a cross-sectional illustration in Figure 13a. Note that this algorithm will construct limiting planes of visibility regardless of whether the ridge line is visible from the given observer, which creates extra work. A better algorithm would consider this before building the limiting planes of visibility:

```
Initial Condition:  A large volume of air above some terrain
    LOOP for Observer IN (all observers)
        LOOP for Ridge-line IN (all ridge-lines)
             IF ridge line visible from Observer THEN
                CONSTRUCT Limiting Plane of Visibility using
                          Observer and Ridge-line
                DIVIDE all volumes at the Limiting Plane of
                       visibility
            END
        END
    END
```

Algorithm 3-2. Better Construction of Visibility Volumes



(a)



(b)

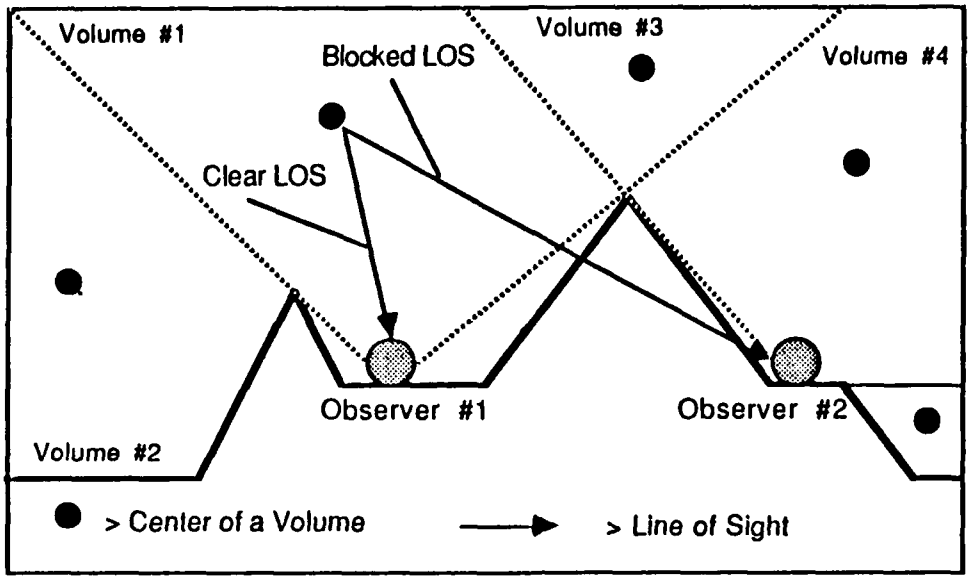
Figure 13. Results of the Visibility-Determination Algorithm

Illustrative results for this algorithm are shown in Figure 13b. This algorithm was not implemented in this thesis.

b. Visibility Criteria

After the visibility volumes are built, their visibility relative to all observers must be determined. This is best done using a simple line-of-sight (LOS) model. Since the visibility volumes define regions of uniform visibility, no point in the volume is more or less visible than any other point. For this reason, visibility can be determined from the center of the volume, which is easy to find, to an observer. If that LOS is blocked by terrain anywhere along its length, then the entire visibility volume is not visible with respect to that particular observer. See Figure 14 for an example.

Once the visibility with respect to all observers has been calculated, the probability-of-detection for all volumes may be calculated in some appropriate manner. This is a function of the individual probabilities-of-detection for all of the observers which "can see" a given visibility volume, and is discussed in detail in the following chapter and in [Ref. 2]. Volumes which are visible to no observer can be assigned a probability-of-detection of zero, or some small, non-zero baseline value.



- Volume #1 is visible from Observer #1 and not visible from Observer #2
- Volume #2 is not visible from Observer #1 and not visible from Observer #2
- Volume #3 is visible to both observers
- Volume #4 is visible to Observer #2 and not visible to Observer #1

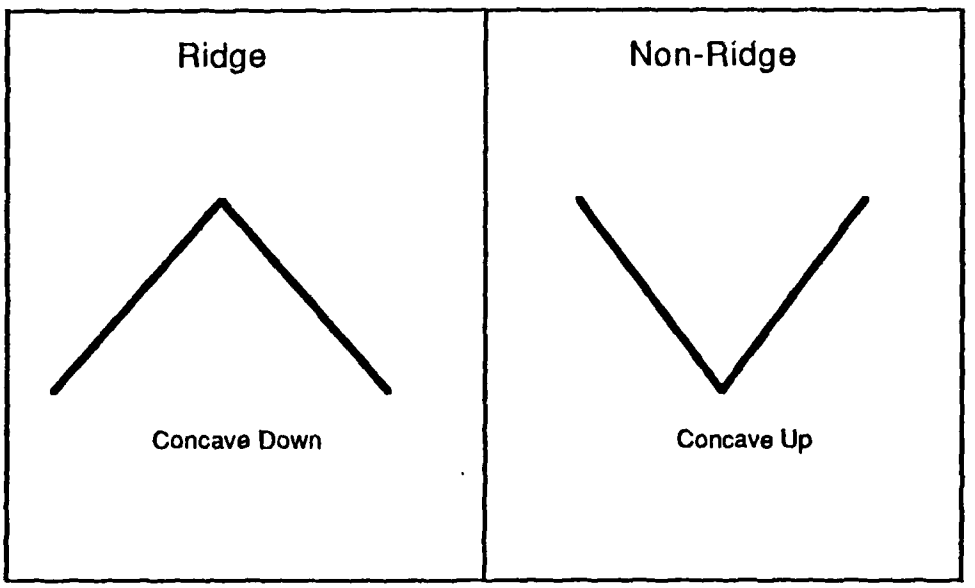
Figure 14. Visibility Criteria

c. Additional Sub-algorithms

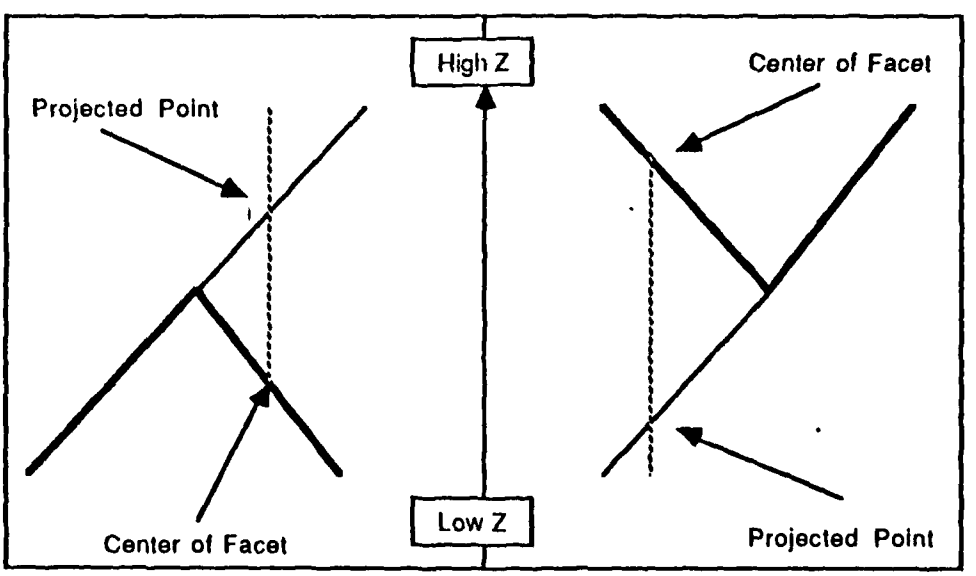
Several other sub-algorithms are required by the static-visibility determination algorithm; input, ridge-finding and graph-building. The input algorithm is highly dependent upon the method of representing the polyhedral terrain model, and its discussion will be delayed until Chapter IV.

(1) Ridge Finding. Implicit in the visibility volume algorithm is the ability to identify ridge lines, information not (generally) available with the input data. A ridge line is a line segment at the intersection of two facets of the terrain model such that the two facets are concave down (see Figure 15a). A complex formula is available to do this in three dimensions [Ref. 12] which is completely general. However, it is possible to take advantage of the polyhedral representation of the terrain to make a much easier algorithm.

First project the center point of one of the facets (it does not matter which) onto the plane of the other facet. If that projection is above the center point (e.g., has a higher altitude, or z value), the line is a ridge line. If any other case arises, then the line is not a ridge line. Figure 15b illustrates this essentially graphical approach. Exactly vertical cliffs are not considered here, since they are assumed not to occur in nature. This also serves to



(a)



(b)

Figure 15. Ridge-Determination Method

avoid identifying structural edges used to bound the terrain model (the top, sides and bottom) as ridge lines.

(2) Graph Building. The earlier steps in the static-visibility algorithm produce a network of adjacent visibility volumes. These volumes will be connected to their immediate neighbors by either a common facet (the usual case), or adjacent, coplanar facets (an unusual, implementation-dependent case). In either event, the connecting facet represents an edge in the search graph, and the centers of the visibility volumes represent nodes in the graph. Visibility volumes are not considered to connect at common volume vertices or volume edges. Building the search graph involves a very straight forward algorithm to find all connections between all visibility volumes:

```
LOOP for volume IN (all visibility volumes)
    LOOP for facet IN (all facets of a volume)
        FIND other volumes CONTAINING facet
            CONCLUDE THAT volume CONNECTS to
                       other volumes
    END
END
```

Algorithm 3-3. Algorithm for Finding Search-Graph Edges

B. PATH PLANNING ALGORITHMS

Our path-planning is a two-step process. First an A* search algorithm is used to find the first optimal volume path from the start volume (the volume containing the start

point) to the goal volume through the search graph. Quantitative factors are included in this first step, such as probability-of-detection and turn cost. This step finds an optimal set of contiguous volumes through which the final path must move. This is the only step in the path planning process in which aerodynamic processes, such as climbs and turns, are explicitly manipulated.

The second step finds an optimal **piecewise-linear path** within the confines of the volume path (see Figure 5, in Chapter I). In order to reduce the complexity of the path planning process, only distance flown and probability-of-detection are optimized, using Snell's Law as an optimizing tool. For reasons to be described later in this section, maneuvers (changes in direction or in angle of climb/dive) are restricted to the boundaries between volumes, so the path produced by this step is a series of linear path segments, joined at the boundaries between volumes.

1. **Volume-oriented A* Search**

The classic A* method has been used in this thesis to find a "reasonable" flight path with a trade-off between minimum cost and minimum probability-of-detection. A reasonable flight path will be defined as a path in which an appropriate amount of effort has been made to avoid regions with a high probability-of-detection, while not going too far off of a minimum cost path. As an example, when driving from Los Angeles, California to Phoenix, Arizona while minimizing

cost and trying to avoid desert terrain, it can be considered reasonable to go via San Diego and Yuma (see Figure 16). It would not be reasonable to go via Barstow and Flagstaff, or via Sacramento and Denver.

As mentioned earlier, heuristics are a key tool in deciding the type of path to be found. [Ref. 2] provides an excellent discussion of the role of heuristics in path planning. Heuristics used here are:

- Remain in volumes with low probabilities-of-detection as long as possible.
- Fly low whenever possible.
- Minimize distance flown.
- Small turns are better than large turns.

These will be implemented as adjustments to the cost and evaluation functions of the A* search. Note that many of these heuristics can conflict with one another, such as the first and third ones. It is up to A* search to resolve these conflicts in a "correct" way.

A* search requires both a cost function and evaluation functions, as described in Chapter II, Section A1. since the A* search is finding a volume path through the search graph, estimators of energy use rather than exact formulas are used. These estimators account for the extra costs incurred when turning or climbing an aircraft. Energy "savings" (e.g., reduced costs over equivalent level flight) can be expected when diving.

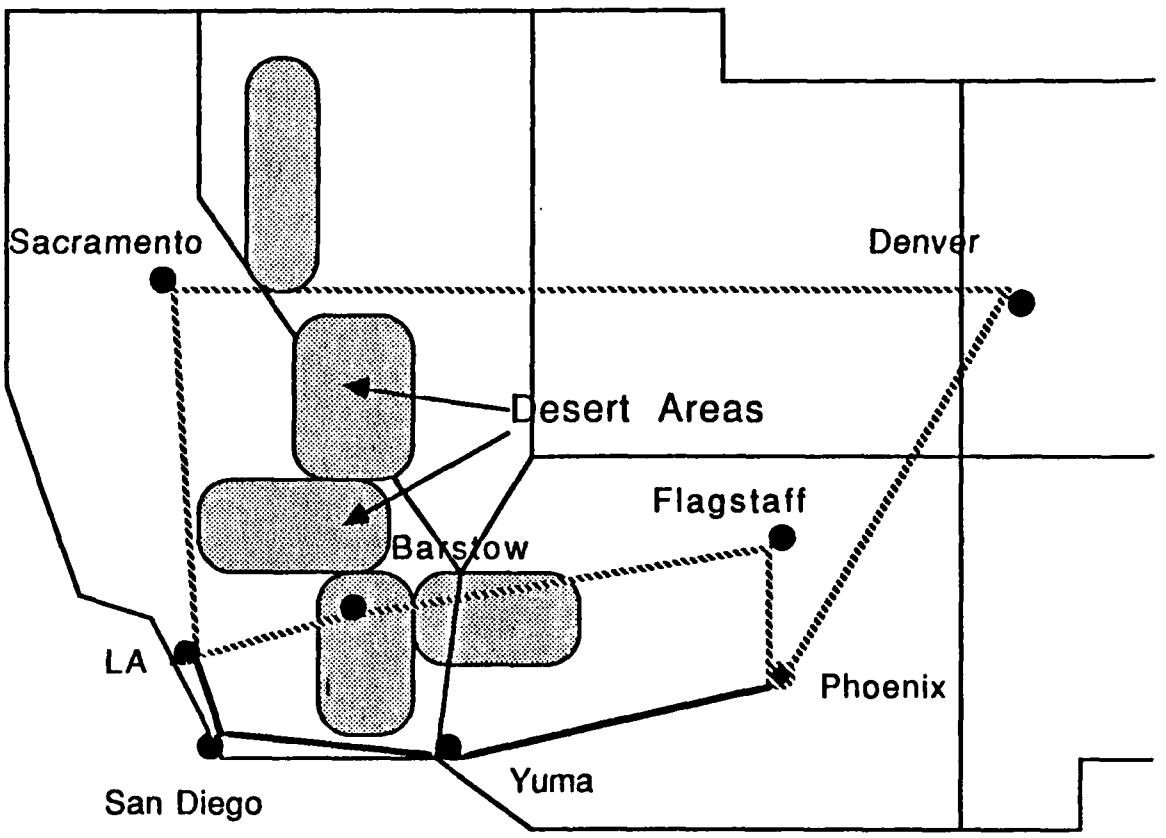


Figure 16. Optimum Routing Example

In order to include non-energy factors in the A* search (such as probability-of-detection), they are used to arithmetically modify the results of the cost and evaluation functions to reflect the "desirability" of being in a region with a given probability-of-detection. Thus regions with a high probability-of-detection will incur a proportionally higher energy use for a given flight path than a similar region with a lower probability-of-detection. This means that no separate decision must be made to select path volumes based solely on probability-of-detection information.

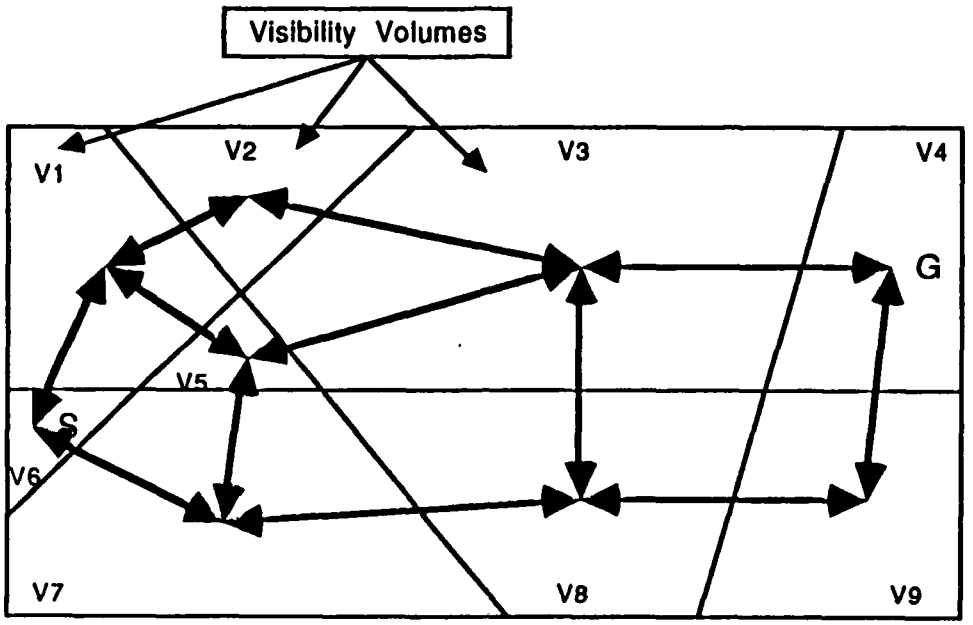
a. Compensated Cost

One approach to figuring costs when using the A* search is to calculate path-segment costs based on volume-center to volume-center distances, as shown in Figure 17. Turn and altitude energy estimators can be calculated from the horizontal and vertical deviation of the current center-to-center path-segment direction from the previous center-to-center path-segment direction.

The modification to the actual cost of a path segment (volume-center to volume-center distance) for probability-of-detection can be expressed as:

$$\text{Net Cost} = (\text{Actual Cost}) * \text{PD-Modifier} \qquad (3\text{-}1)$$

where "PD-Modifier" is some function of the probability-of-detection.



- Start Point at "S"
- Goal Point at "G"
- Visibility Volumes named "V1" through "V9"
- Volume center-to-center paths marked by ◄——►

Figure 17. Volume Center Paths

For a flying object, certain maneuvers represent a certain fixed cost. A turn will incur such a fixed cost, which is a function of the speed of the flying object and the degree of turn. Increased turn angles will incur increased fixed turn costs, up to some maximum allowed turn angle. This can be repeated in simplified form by modifying Equation 3-1 as follows:

Net Cost=(Actual Cost + Turn Cost) * PD-Modifier (3-2)

Finally, flying objects use extra energy when climbing, at a rate which is a function of the length of the climb (the Actual Cost) and the degree of the climb. They also use less energy when diving, again as a function of the length and steepness of the dive. A final change must then be made to Equation 3-1 which represents a modification for altitude changes. Note that this modifier must be a multiplicative modifier, since the energy gain/loss is a function of the length of the climb or dive.

Net Cost=(Actual Cost + Turn) *
Altitude-Modifier * PD-Modifier (3-3)

Equation 3-3 represents the total compensated cost for a path segment. It can be used when calculating both the cost and evaluation functions during the A* search.

b. Search Algorithm

The classic A* search algorithm, as modified for this thesis, begins with an agenda initialized to all

successors (adjacent volumes) of the volume containing the start point, and all cost and evaluation functions precalculated for those successor volumes. Then:

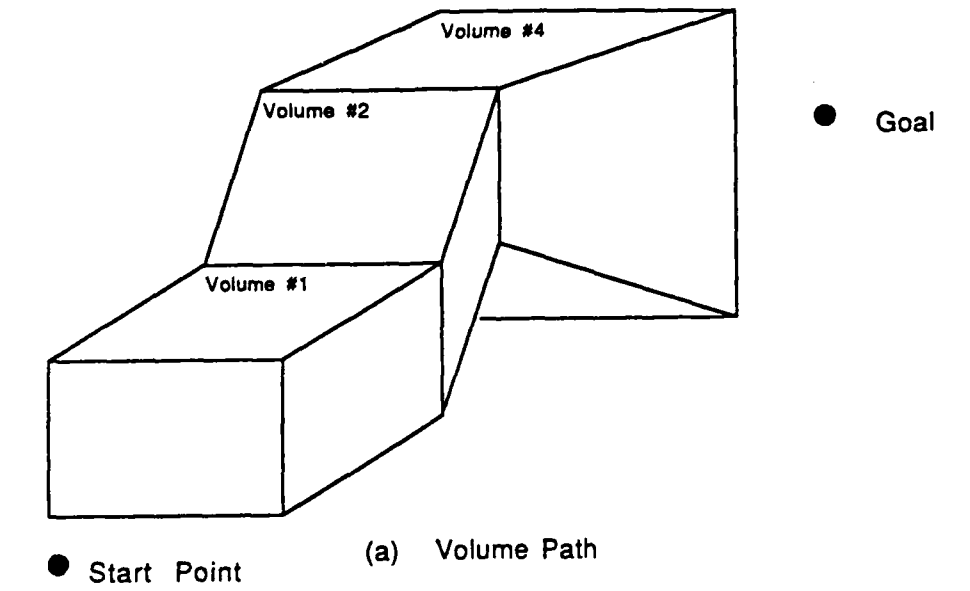
```
LOOP UNTIL goal-volume is on the agenda
    REMOVE the best successor from the agenda
    FIND all successor volumes to the best successor
    UPDATE path information for the successor volumes
    CALCULATE cost and evaluation functions for the
              successor volumes
    INSERT all successor volumes onto the agenda
END
```

Algorithm 3-4. Search Algorithm for Finding Volume Path

The A* evaluation function calculates the compensated cost for the distance from the center of the successor volume to the goal, ignoring obstructions, and assuming that the probability-of-detection of the successor volume applies to the entire distance to the goal. Note that this evaluation function is not the lower-bound evaluation function. The cost function is a simple summation of all compensated costs incurred up to the current volume along the current path. Adding these cost and evaluation functions gives the total cost for that successor volume

c. Optimal Piecewise-Linear Path

The optimal piecewise-linear path is the minimum-cost path, consistent with the objective of maintaining the lowest possible probability-of-detection along the path, which lies entirely within the volume path (see Figure 18).



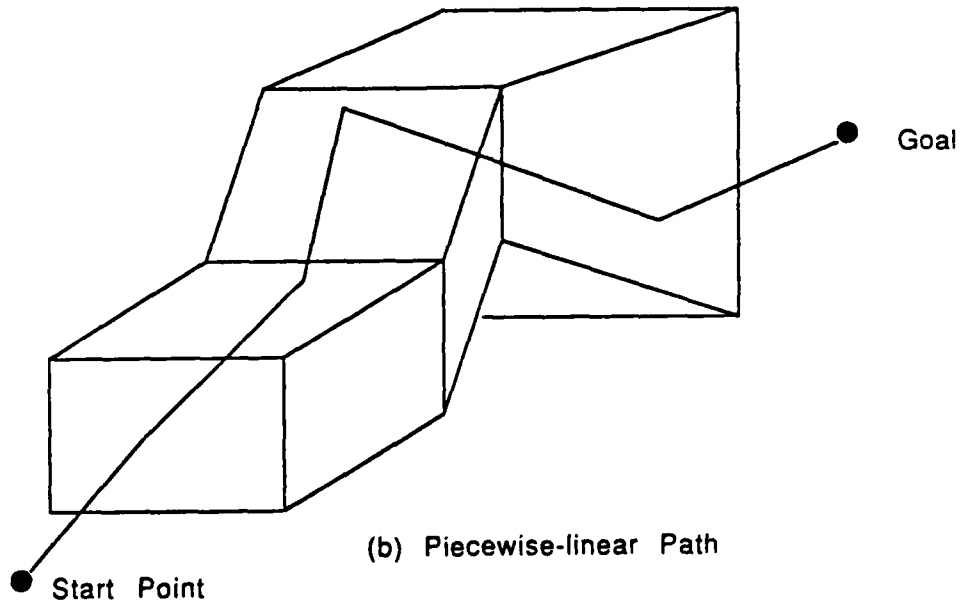


Figure 18. Types of Paths

In order to reduce the complexity of the path planning process, this minimum cost is not modified in any way by the maneuvers along the flight path, but is simply the sum of the point-to-point distances along each path-segment of the total path. The only constraints placed on the linear path are that it must be contained entirely within the volumes comprising the volume path (since this was determined to be the optimal way to go), that maneuvers occur only at the edges of volumes, and that the distances traveled in volumes with high probabilities-of-detection be minimized.

The first two constraints can be met if all visibility volumes are convex, and all linear path line segments are "built" between facets. The convex structure of visibility volumes will ensure that any line segments connecting facets of the volume will always lie completely inside the volume [Ref. 13]. Ensuring that maneuvers occur only at volume facets is an implementation issue.

At this point, it may be useful for the reader to discard the concept of visibility volumes, and consider the construction of an optimal linear flight path as a situation in which a flight path must be found from a start point through a series of restrictive **windows** (see Figure 19) to the goal [Ref. 14]. The windows, in this case, are the facets which connect the volumes of the volume path. Probability-of-detection information must still reside in the spaces between the windows. These windows, or facets, will

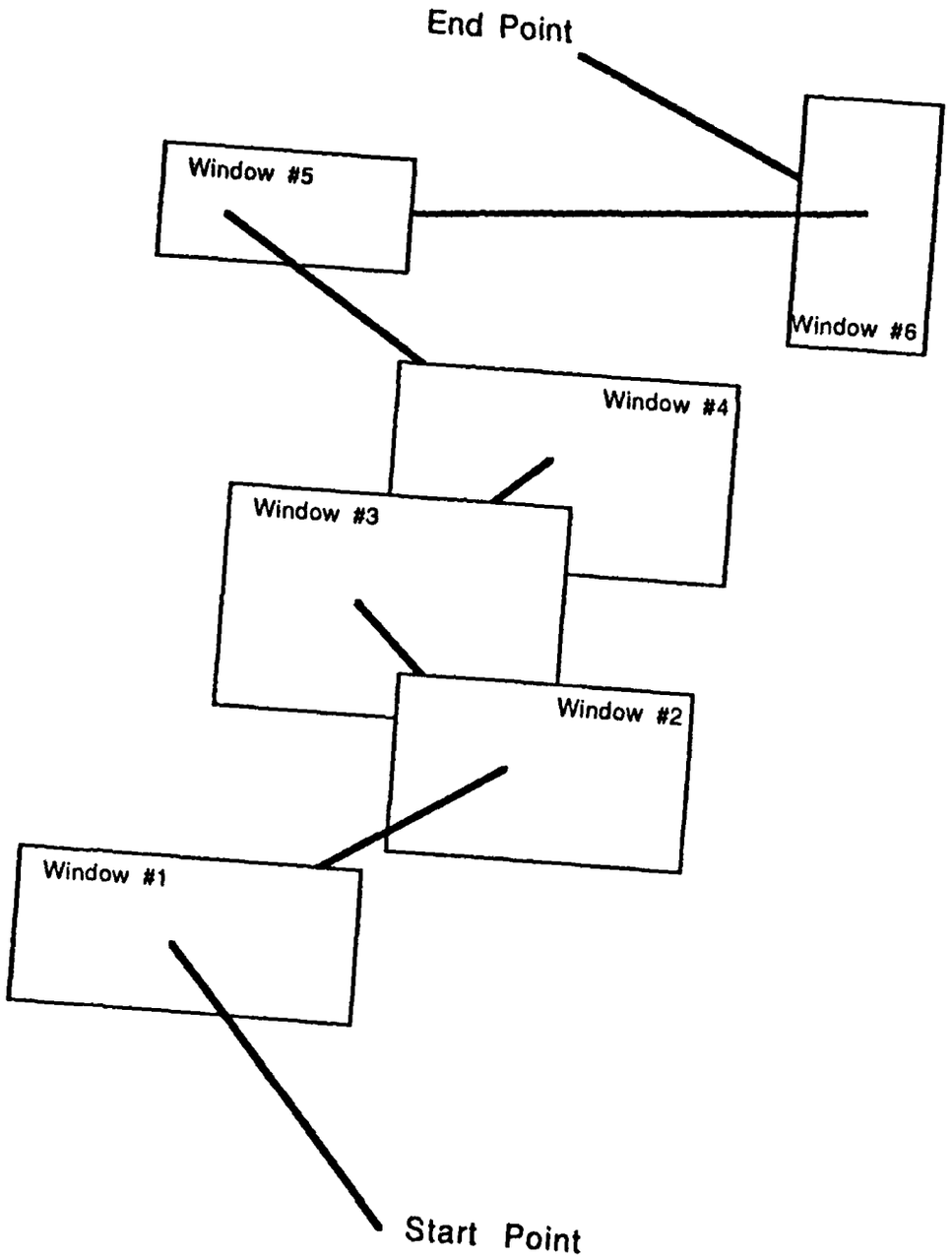


Figure 19. Path Through a Series of "Windows"

be called **maneuver facets**, reinforcing the requirement that all maneuvers must be conducted within the plane of these facets.

Before proceeding, an initial guess at the piecewise-linear optimal path is needed. Any suitable choice will do, so a very general maneuver-facet-center to maneuver-facet-center path can be selected as the initial guess.

The third constraint, the need to minimize the distance traveled through high probability-of-detection volumes, can be handled by applying an adaption of Snell's Law. Using the probability-of-detection of a volume as the index of refraction for Snell's Law, the angles of incidence, and refraction as shown in Figure 20, Snell's Law will be:

$$(Pd1 + 1)*\sin ß1=(Pd2 + 1)*\sin ß2 \qquad (3\text{-}4)$$

or

$$(Pd1 + 1)*\sin ß1 - (Pd2 + 1)*\sin ß2 = 0 \qquad (3\text{-}5)$$

or, if Snell's law is not exactly met,

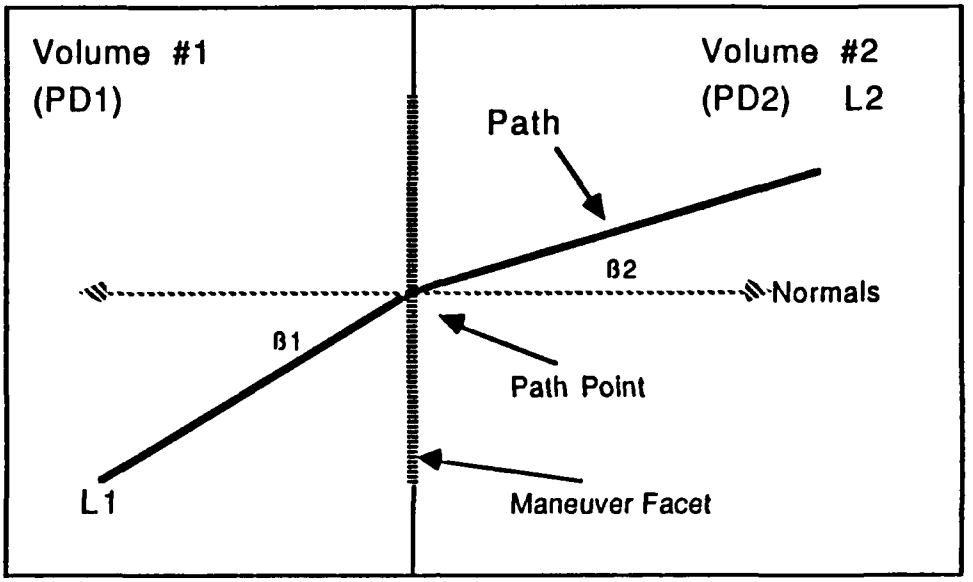
$$(Pd1 + 1)*\sin ß1 - (Pd2 + 1)*\sin ß2 = e \qquad (3\text{-}6)$$

where

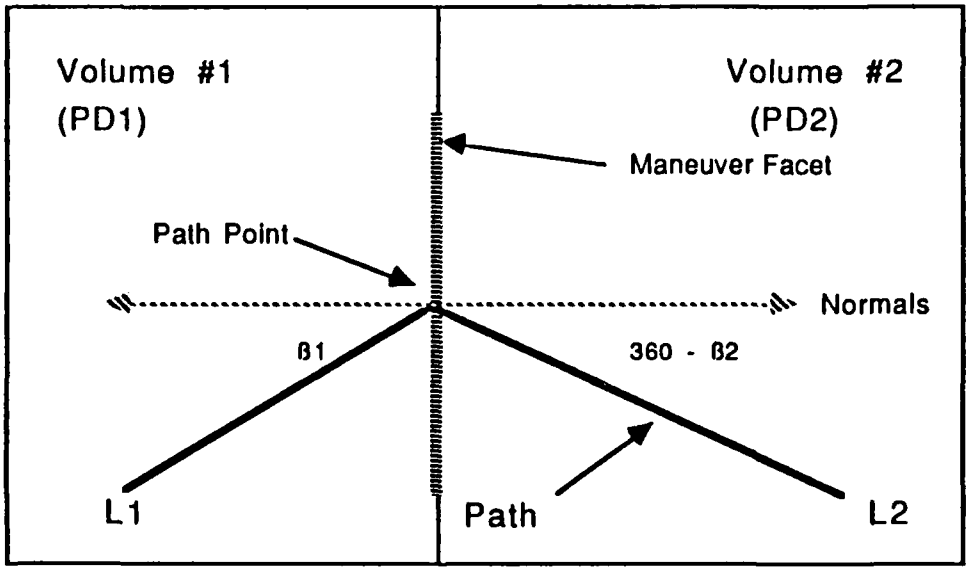
Pd1 is the probability-of-detection on the incident side,

Pd2 is the probability-of-detection on the refracted side,

ß1 is the incident angle,



(a)



(b)

Figure 20. Snell's Law Optimization Situations

ß2 is the refracted angle, and

e is an error item.

The index-of-refraction of Equation 2-1 is represented by one plus the probability-of-detection. This is simply to avoid having an index-of-refraction of zero when Pd1 or Pd2 is zero. Note that the two coefficients (Pd1+1) and (Pd2+1) are constant, and that ß1 and ß2 can be varied by moving the path point on the maneuver facet (see Figure 20), assuming that the other ends of the path lines remain fixed. Note also that ß1 and ß2 are actually functions of the position that the path intersects the maneuver facet, called the path point or maneuver point. As the maneuver point is moved, the values of ß1 and ß2 will change, and the value of the error in Equation 3-6 will also change. Only when Snell's Law is exactly satisfied will Equations 3-5 and 3-6 be equal. Figure 20b does not correspond to a situation found in optics, but such an arrangement does occur in path planning.

If the maneuver point is constrained to move along a line within the maneuver facet (only), then Snell's Law will guarantee that a minimum value for the error in Equation 3-6 can be found ([Ref. 3] and [Ref. 4]). If one notes that the path segments L1 and L2 of Figure 20 must lie in a plane, then the movement of the maneuver point can be constrained to lie in the line defined by the intersection of the plane of the maneuver facet and the plane formed by L1

and L2. Further restricting this line to be a line segment contained within the maneuver facet will ensure that a minimum value for the error term in Equation 3-6 can be found within the maneuver facet.

Finding the optimal linear path, then, involves applying the above Snell's-Law optimization to each maneuver facet along the volume path in turn, keeping all other maneuver points fixed. A single optimization pass occurs when all maneuver points have been adjusted one. The optimization passes must then be repeated until the length of the entire linear path converges to some optimal value.

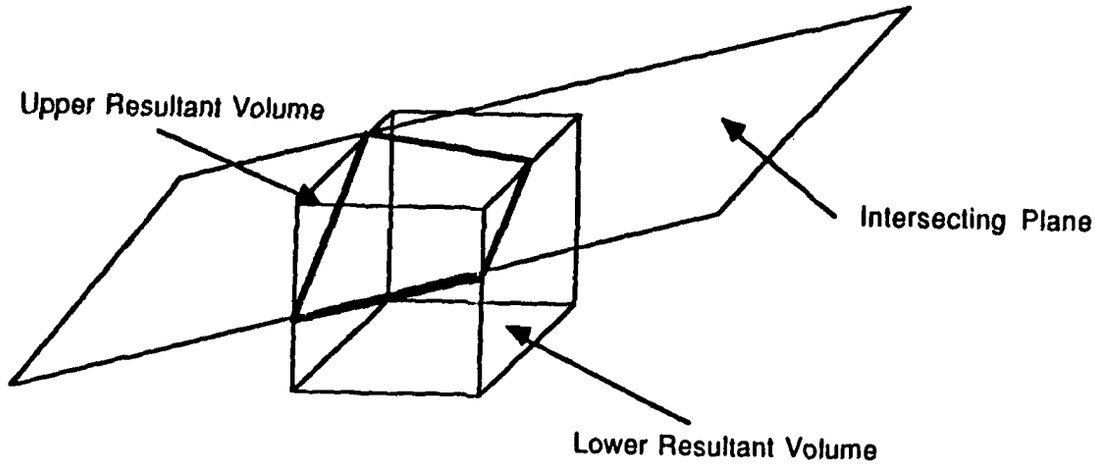
C. SUPPORT ALGORITHMS

1. Intersection of a Plane and a Volume in Space

The intersection of a plane with an arbitrary polyhedral (planar) volume in three-dimensional space depends on the shape of the volume. For the purposes of this discussion, volumes may be classified into two general types: convex and non-convex planar volumes.

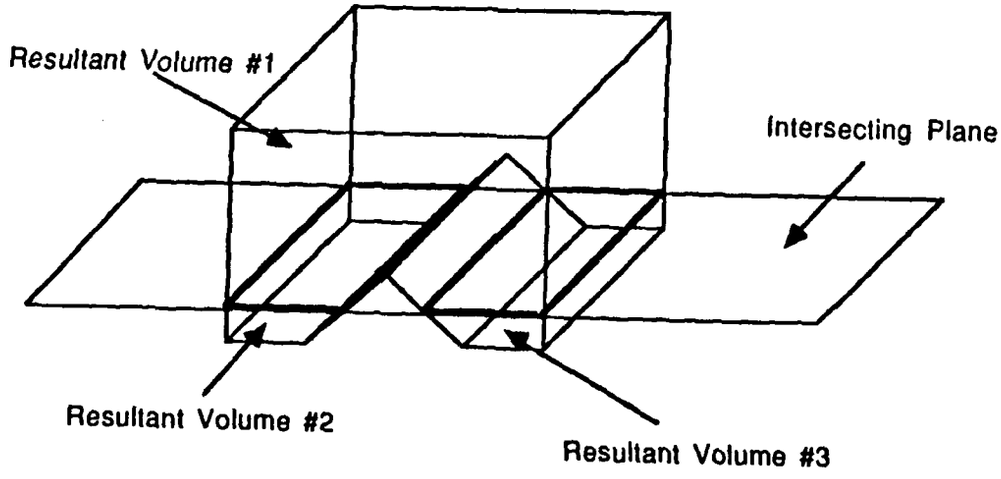
a. Interception with Convex Volumes

Figure 21a shows that the interception of a convex volume will create two smaller resultant volumes. These resultant volumes are composed of whole and partial edges from the original volume plus new edges created by the intersection process itself. The newly created edges always lie in the intercepting plane, and in a plane of one of the other facets of the original volume. In general, the facets



Simple Convex Volume Intersection

(a)



Simple Non-Convex Volume Intersection

(b)

Figure 21. Volume Intersections

of the original volume will not be unaffected by the interception; several will be subdivided by the process.

There are several "degenerate" cases of interception shown in Figure 22. Note that the original volume lies entirely on one side of the intersecting plane. This observation will be useful in detecting these cases.

b. Alternative Algorithms

While several methods have been presented to intersect a plane with a volume [Ref. 10], two approaches immediately suggest themselves. The first finds the lines defined by the intersection of the plane and the planes of the facets of the volume, and then fits these "lines of intersection" onto the facets. The second approach finds the points defined by the intersection of each edge of the volume with the intersecting plane, and connects these points along the facets of the volume.

The first approach was not used due to the complexity of the fitting procedure. The second approach, the "plane-edge method," involves a simpler intersection of edges and a plane. Each edge of the original volume is intercepted with the intersecting plane in turn, possibly subdivided according to the results of the interception, and then placed in the appropriate resultant volume. Edges that lie in the plane of the interception are found or built by connecting the intersection points of the edges and the plane in an appropriate manner, and then placed in both resultant

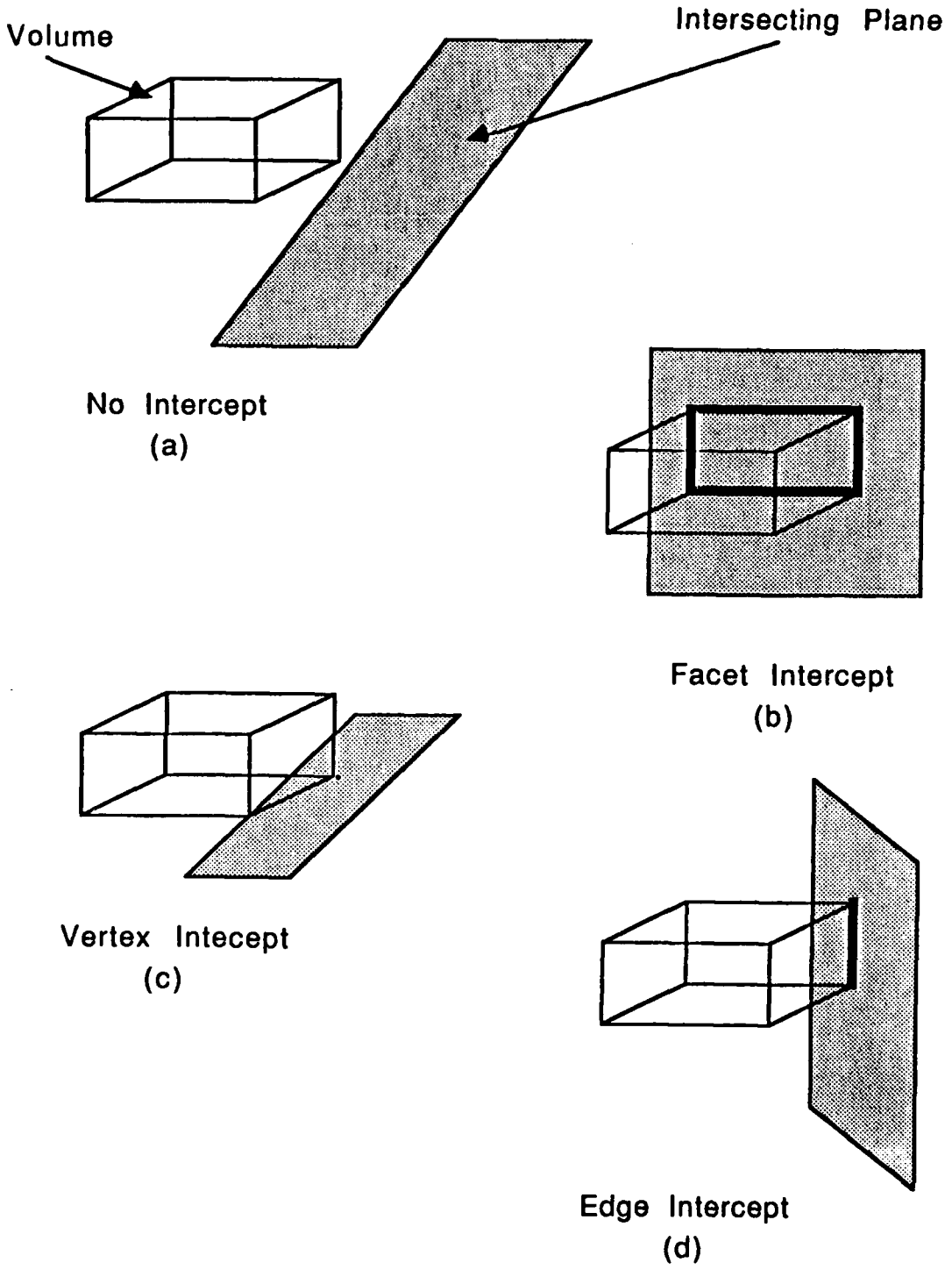


Figure 22. Degenerate Intercept Cases

volumes. Finally the resultant volumes are checked for the degenerate conditions that indicate that a partial intercept has occurred.

(1) Intersection of a Line and a Plane. The intersection point of a generalized line and a plane in three space is derived from the equation of a plane [Ref. 11]

$$ax + by + cz = Ao \qquad (3\text{-}7)$$

and the vector-parametric representation for a edge described in Chapter II [Ref. 11]

$$r(t) = Ai + Bj + Ck + t(Di + Ej + Fk) \qquad (3\text{-}8)$$

where

Ai + Bj + Ck is the equation of the position vector.

Di + Ej + Fk is the equation of the direction vector.

t = 0 at one end-point of the edge and

t = a maximum value at the other end-point of the edge (called "t-max").

As a result, all points on the line are given by the parametric equations

$$x = A + tD \qquad (3\text{-}9)$$

$$y = B + tE \qquad (3\text{-}10)$$

$$z = C + tF \qquad (3\text{-}11)$$

Solving these equations simultaneously yields

$$t = [Ao - (aA + bB + cC)]/[aD + bE + cF] \qquad (3\text{-}15)$$

at the intercept point, which is on the line defined in Equation 3-8.

(2) Intersection of a Edge and a Plane. Intercepts between a edge and a plane may result in one of three situations:

- $0 \leq t \leq$ t-max: This indicates that the intercept occurred within the edge. If t equals either 0 or t-max, then the intercept occurs at one of the endpoints, otherwise it occurs at some middle point along the edge.
- $t < 0$ or $t >$ t-max: This indicates that the intercept has occurred beyond the endpoints of the edge. No intercept occurs.
- t is infinite: This occurs when the denominator of Equation 3-15 equal zero. In this case, the edge is parallel to the intersecting plane. Again, no intercept occurs.

(3) Building the Resultant Volumes. Using the assumption that two new volumes will be created upon intersection of a convex volume by a plane, the results of the line-plane intersection described above can be used to create the resultant volumes as the process occurs for each edge of the volume. In the case that no intercept occurs for a particular edge (the second and third cases above) one point of the edge must be checked for its location relative to the intersecting plane, then added to the appropriate resultant volume. For an intercept at an endpoint of an

edge, the other endpoint must be checked for its location relative to the intersecting plane, and then the unchanged edge may be placed in the appropriate resultant volume.

When the plane intersects the edge anywhere in its middle, the edge must be divided into two smaller edges. These may be placed into the resultant volumes using the procedure used for an intersection at an end point.

New edges must be built in the plane of the intercepting plane. Note that any lines or points in this plane belong in both resultant volumes. The algorithm must find the edges which connect the intercept points in this region, within the constraints of the particular construction of the original volume. These new edges demark the effect of the intersecting plane passing through the original volume, and construction of these edges involves connection of interception points in the planes of the facets of the original volume.

Now it is possible to check for the degenerate intercept conditions discussed earlier. If an intercept occurred then both of the resultant volumes must not be degenerate volumes. To be a valid volume (non-degenerate) several independent minimum conditions must be met. The resultant volumes must

- have at least four points
- have at least six edges
- have at least four facets

- meet Euler's Relation [Ref. 15],

$$V + R - E = 2 \qquad (3\text{-}15)$$

where

E is the number of edges in the volume,

V is the number of vertices in the volume, and

R is the number of regions (facets) in the volume.

If either of the resultant volumes does not meet any of these degeneracy checks, then the original volume is unpartioned by the intersection process.

c. **Intersection with Non-Convex Volumes**

A non-convex planar volume is one with at least one non-convex surface feature. Intersection of such a volume with a plane will produce at least two resultant volumes, but often more (see Figure 21b). It is this inability to determine exactly how many resultant volumes which will be produced, and the inability to predict their location relative to the intersecting plane, that makes interception of this type of volume difficult.

Fortunately, only one subcase of this problem arises when dealing with polyhedral terrain models. This is illustrated in Figure 23. This special situation is easily identified by adding a step to the intersection algorithm for all new edges created in the plane of the intercepting plane that tests that some non-endpoint point on the new edge (such

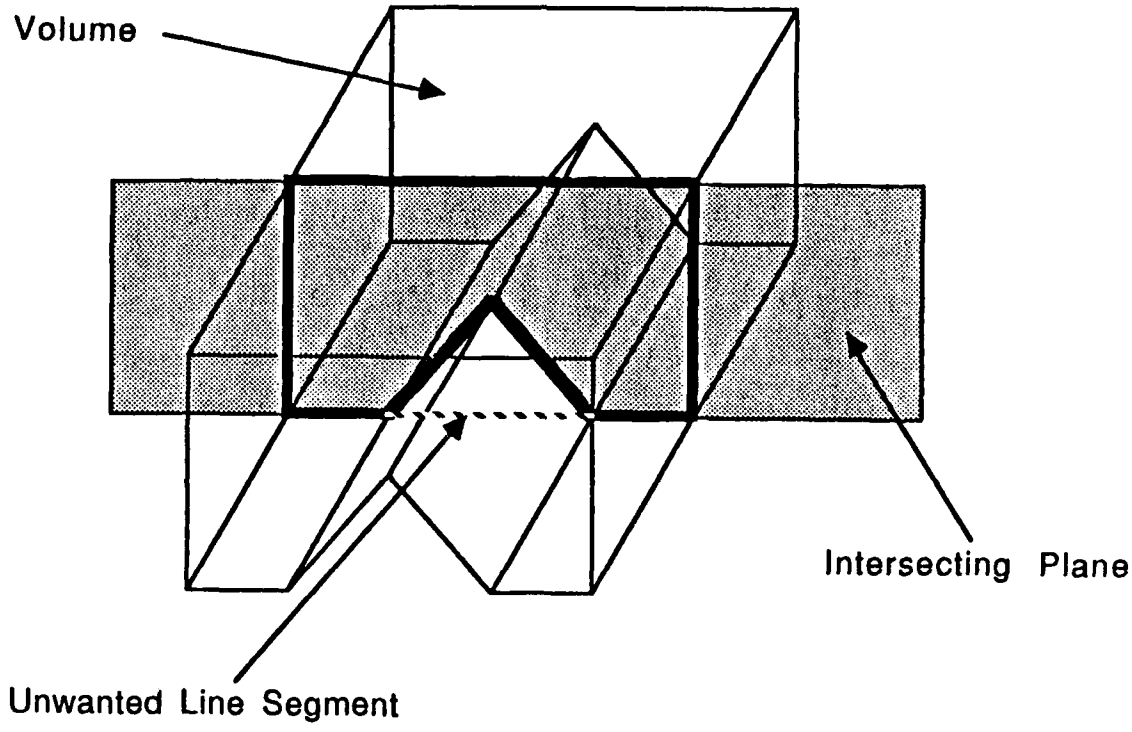


Figure 23. Non-Convex Intercept

as a midpoint) to see if it is included inside the original volume. If this boolean test is true, the new edge is "legitimate." If the test is false, then the new edge is outside the original volume, and must be discarded. Only one point needs to be tested for inclusion in the original volume, since the unwanted edges lie entirely outside the original volume (except for their endpoints, of course).

The non-convex interception case only occurs during an early part of the static visibility-determination phase, and then only during a clearly identifiable portion of the algorithm. This is discussed in more detail in the following chapter.

2. **Facet Detection**

Recall that a facet is composed of a set of planar points connected in a cycle by edges. The intersection algorithm places points and edges into the appropriate resultant volumes, but no facet information is passed from the original volume to the resultant volumes. The first of the two facet finding methods presented here needs only this information to find all of the facets of the resultant volumes; the algorithm searches through the edges of each resultant volume to find all facets. This is the **search-based facet finding method.** The second method presented cheats a little, and uses some information from the original volume to help find the facets of the resultant volumes after

an interception. This is the plane-based facet finding method.

a. Search-Based Facet Finding Method

This algorithm uses a best-first search to find all the facets of a volume. Recall that a best-first search is an A* search which uses only an evaluation function. This facet-determination search finds a plane in a volume defined by two edges, and then attempts to build the shortest possible coplanar closed loop of edges back to a designated endpoint of one of the lines.

The search is initialized by choosing any point as a start point, and then finding two connected edges which have the start point as an endpoint (as edges L1 and L2 of Figure 24). New edges are added to this initial set of lines if and only if:

- The new edge is connected to the end of the current edge, and
- The new edge is in the same plane as the initial two edges, and
- The opposite endpoint of the new edge is closer to the start point than any other edge which meets the criteria above. This causes the search to loop back towards the start point, creating the needed cycle of edges.

In the volume of Figure 24, for example, the search will produce a facet consisting of edges L1, L2, L4, L6, L8 and L9, all endpoints of those edges, and the plane equation of the shaded plane.

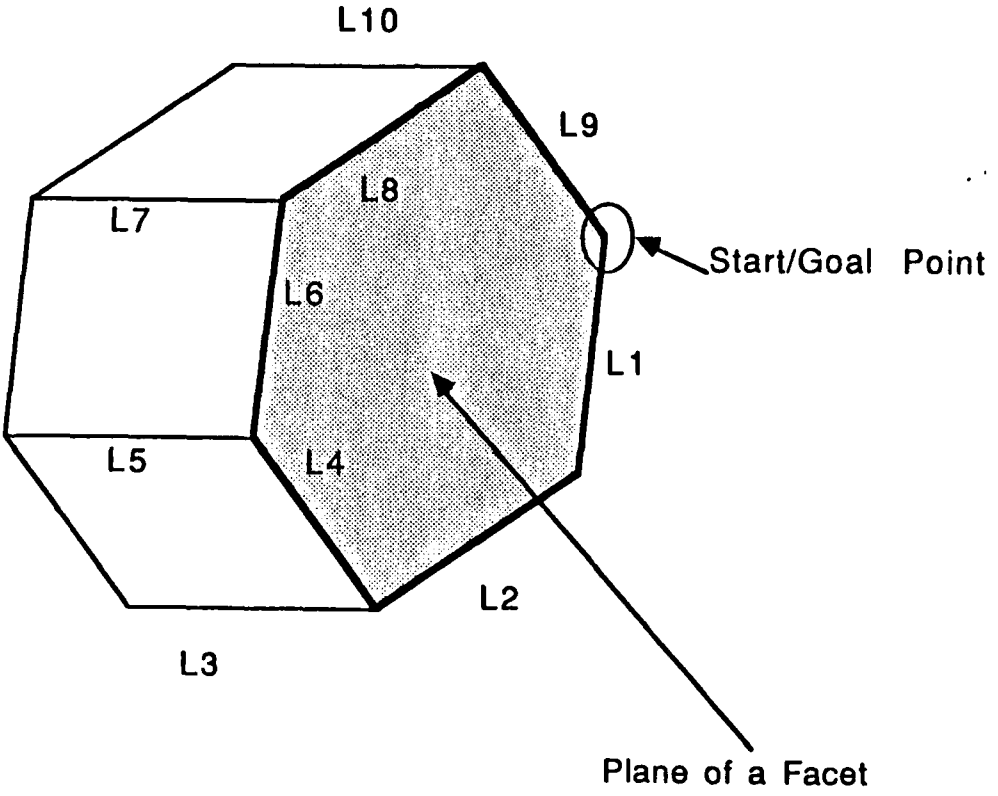


Figure 24. Facet-Determination Example

b. The Plane-Based Facet Finding Method

This method relies on the fact that all of the facet plane equations of the original volume are known, as is the plane equation of the intersecting plane. Since the resultant volumes are made from the elements of the original volume, all of the facets of the resultant volumes will have plane equations equal to those of the original volume, plus one with the plane equation of the intersecting plane. Some of the original volumes' plane equations may appear in only one resultant volume, and some will appear in both resultant volumes.

As a result, to find all of the facets of the resultant volumes, points and edges need only be partitioned into the plane equations in which they lie. Points and edges will lie in only two plane equations. Facets may then be determined relatively quickly and easily.

This method requires that the original volume and resultant volumes be convex, however, since no attempt is made to establish the connectivity of line segments in the facets created; this is assumed to be true.

c. Discussion

As can be inferred from the above description, the facet determination methods each have certain cases in which they are best used, and others in which they are best avoided. In general, the search-based method is suitable for most types of non-convex volumes, and when inputting terrain

data. The plane-based approach is best used for all types of convex volumes where some previous information is available. Both algorithms were implemented. The search-based method is used when dealing with non-convex volumes and after certain error conditions. The plane-based method is used for all convex volumes.

IV. IMPLEMENTATION

The algorithms described in the previous chapter were implemented in LISP. LISP was selected due to the availability of LISP machines and advantages in speed, numerical accuracy and sophisticated data-structure management.

A. SYSTEM SPECIFICATIONS AND REQUIREMENTS

The program was implemented on a Texas Instruments (TI) Explorer II LISP machine with 16 megabytes of memory, 60 megabytes of virtual memory, using Common LISP with the LISP Flavor System (version 6.0). The LISP code will also run on a TI Explorer I LISP machine with significant increases in execution times. The LISP code should also run on any Symbolics LISP machine with only minor changes to the flavor declarations and message passing formats, although this has not been tried.

The LISP implementation is in the seven files shown in Table 2. All primary data structures were written using the LISP Flavor System, an object-oriented programming environment. Some minor data types were implemented as property lists. The uncompiled code (interpretable) in the seven files occupies about 150,000 bytes (150K) of memory, and the compiled version occupies approximately 140K of memory.

TABLE 2. DISK FILES FOR THREE-DIMENSIONAL PATH PLANNING

Filename	Size (text/compiled) (Kbytes)	Description
SETUP.LISP	16/8	Main control function for most of static visibility determination phase. Includes terrain input functions.
VOLUME-FLAVORS.LISP	31/50	Defines all flavors used in the visibility determination phase, and their associated methods and support functions
COMMON-FUNCTIONS.LISP	25/15	LISP functions used by all other functions in the implementation.

TABLE 2 (CONTINUED)

Filename	Size (text/compiled) (Kbytes)	Description
INTERCEPT.LISP	15/9	Functions to intercept a volume and a plane.
VISIBILITY-FUNCTIONS.LISP	24/15	Controls creating of the visibility regions, and builds the search lattice.
PATH-PLANNING.LISP	29/27	Functions related to path planning.
CAMERA.LISP	15/12	Graphic display functions.

1. Terrain Input Requirements

Two alternate terrain input data formats have been implemented. The first method, accepts terrain data as a series of disjoint edges defined by endpoints. All structural lines shown in Figure 25 (and defined in Chapter III) must be included in the input data. The second input format accepts terrain data as a series of disjoint facets defined by the vertices of the facet. This input method will create the structural lines, with certain restrictions.

The first input method is designed as a simple, fast method of entering terrain data. Input is volume-oriented, and only two volumes may be entered: one ground volume and one air volume. The input format is shown in Table 3. Certain non-convex terrain features may not be entered using this format, including peaks and other embedded terrain features (see Figure 26).

The second input method is much more general than the first, and is what "real" polyhedral input terrain would be like. The method inputs only the terrain itself, builds all structural lines, and creates the air and ground volumes. Terrain may be input in multiple rectangular segments, which allows for entering the embedded terrain features of Figure 26. The only restrictions on the terrain data are that the terrain segments must be rectangular with sides parallel to the x-axis and y-axis. Table 4 provides an example of the data format for this input method.

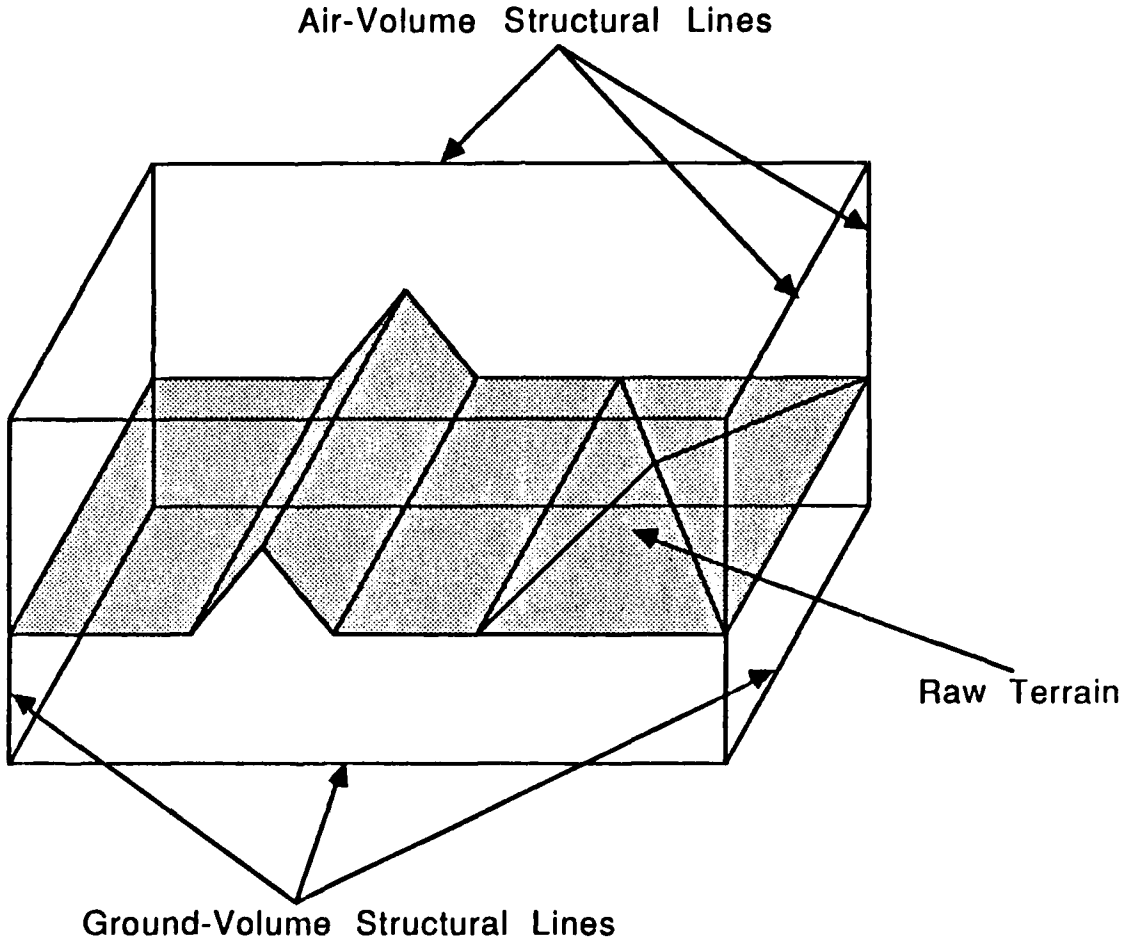
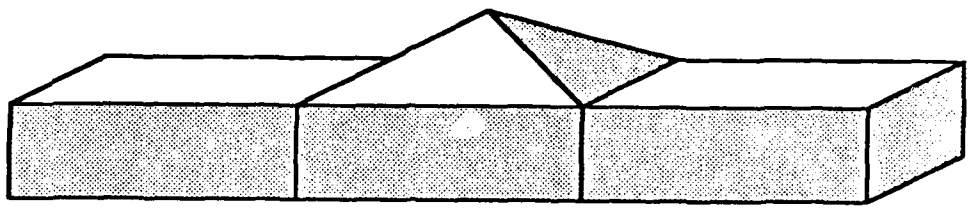


Figure 25. Terrain Structure

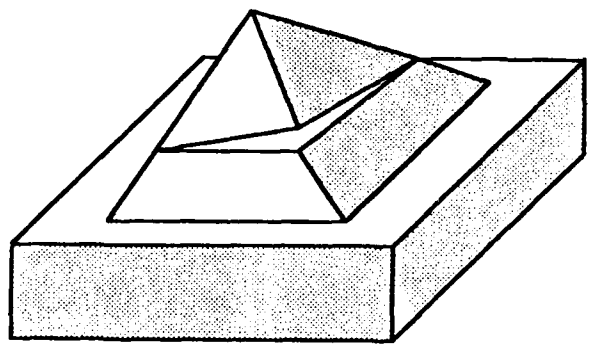
TABLE 3. FIRST INPUT METHOD FORMAT

Type of Data	Format	Example
Point	(x y z)	(10 10 10)
Line	((start-point) (end-point))	((0 0 0) (10 10 10))
Volume	((Line-1) (Line-2)...(Line-n))	(((0 0 0) (10 10 10)) ((10 10 10) (20 20 20)) ((0 0 20) (100 35 15)) ..... ((100 0 100) (100 100 100)))



Peak

(a)



Embedded Terrain

(b)

Figure 26. Difficult Terrain Features

TABLE 4. SECOND INPUT METHOD FORMAT

Type of Data	Format	Example
Point	(x y z)	(10 10 10)
Facet	((point-1) (point-2) (point-3))	((0 0 0) (10 10 10) (0 0 20))
Terrain	((facet-1) (facet-2)..(facet-n))	(((0 0 0) (10 10 10) (0 0 20)) ..... ((0 0 0) (20 20 20) (20 0 0)))

Neither input method is fault tolerant. Data is assumed to be correct, with no missing facets, points or edges. Sample terrain data for both formats is included in Appendix A.

2. Output Formats

Two outputs are produced by the implementation, text and graphical. The text output shows the state of execution, while the graphical displays show the results of computations.

Graphic displays are two-dimensional true-perspective projections of wire-frame images of various terrain, volume or path features. No shading or hidden-line algorithms are used. The simple graphics flavors used were not written as a part of this thesis [Ref. 16]. Various fixed viewpoints can be used to display objects in one display window. Multiple display windows are shown at once. Alphanumeric representations showing locations of observers, start and goal points.

Graphic displays include:

- (DISPLAY): Display all volumes.
- (DISPLAY-VOLUMES {List}): Display selected volumes.
- (DISPLAY-VISIBLE {Observer}): Display volumes visible from a given observer.
- (DISPLAY-NOT-VISIBLE {Observer}): Display volumes not visible from a given observer.
- (DISPLAY-PATH {Path}): Display a given path and all ground volumes.

- (DISPLAY-PATHS {List}): Display selected paths and all ground volumes.

Examples of most of these displays are shown in Chapter V.

B. DATA STRUCTURES

The data structures used in the three-dimensional path planning program closely mirror the basic geometric concepts presented in Chapter II. Information relative to path planning, visibility and other important parameters has also been added.

The LISP Flavor System defines templates for data structures (called flavors), which can then be created (instantiated) with unique names and components (called instantiation values). LISP flavors and instantiation values are equivalent to frames and slots as defined in [Ref. 2] and [Ref. 17]. These instantiations (or frames) can represent points, vectors, edges, facets, planes, volumes, paths or other objects. All major data structures in this implementation were written using flavors.

Volumes and paths are the basic units of manipulation in the three-dimensional path planning program, but a higher level exists. That higher level is the "universe," the collection of all observers, volumes (ground and air) relative to a particular terrain model. As the visibility determination progresses, all the changes in volumes are made

within the context of this universe flavor. The names of all paths are kept in a single global variable *LIST-OF-PATHS*.

Volumes, as discussed in Chapter II, are composed of points, edges and facets, as well as many other data items. Table 5 shows all of the instantiation values (slots) in the volume flavor. Similarly, each of the main constituent parts of a volume are themselves flavors, and have other, unique data structures, as shown in Table 6. As described in Chapter II, edges are composed of vectors. this flavor is also described in Table 6.

The remaining flavors used are for observers and agenda-items (for the A* search). Details concerning these flavors may be found in Appendix B.

C. PROGRAM STRUCTURE

The path-planning program presented here has a sequential structure which follows from the algorithms presented in Chapter III. an annotated black-box control diagram is shown in Figure 27.

1. Visibility-Model Functional Description

Functions SET-UP and SET-UP-2 implement the set of static-visibility determination algorithms described in Chapter III. Other than starting these functions by calling them, and entering some observer data between the functions, little other user intervention is required. Upon completion of the visibility determination phase, the user may use the DISPLAY or DISPLAY-VOLUMES function to verify the results.

TABLE 5. SLOTS IN THE VOLUME FLAVOR

Instance Variable/Slot	Description
Visibility	A list of all observers with an unobstructed LOS to the volume.
Probability of Detection	The combined probability-of-detection for all observers in the visibility slot.
Composition	Ground or air (material in volume).
Points	A list of all the points (vertices) of the volume.
Edges	A list of the names of all the line-segments (edges) of the volume.
Facets	A list of all the facets (faces) of the volume.
Arithmetic Center	The point at the center of the volume (used for LOS and path planning calculations).
Connected to	A list of all adjacent visibility volumes.

Mixin Flavor: Graphic (used for display of the volume). See Appendix B.

TABLE 6. SLOTS IN OTHER FLAVORS

Instance Variable/Slot	Description
x-coord	X coordinate value (a number)
y-coord	Y coordinate value (a number)
z-coord	Z coordinate value (a number)

(a) Point Flavor

i	X coefficient of the vector
j	Y coefficient of the vector
k	Z coefficient of the vector
Start-point	Name of the vector start-point
End-point	Name of the vector end point

(b) Vector Flavor

t-max	Parameter value at end of line-segment
Position vector	Name of the position vector
Direction vector	Name of the direction vector
Characteristics	Ridge or nil (not a ridge)

(c) Line-segment Flavor

Edges	A list of the names of all of the line-segments of the facet.
Composition	Not used

(d) Facet Flavor

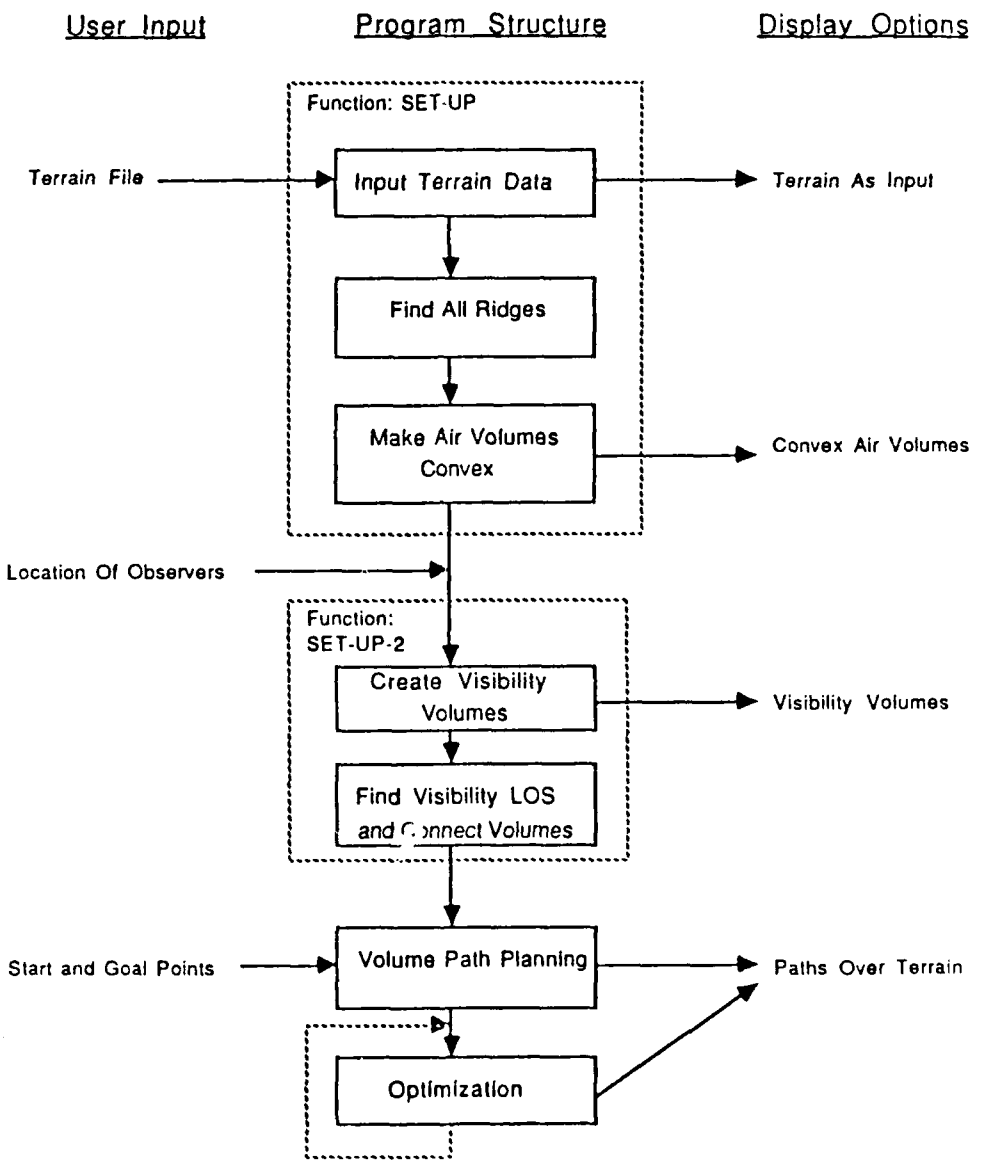


Figure 27. Block Diagram of Program Structure

Several global variables are used to save intermediate states of the static-visibility determination process. These variables are shown in Table 7.

a. Input Terrain Data

All constants are initialized and global variables reset when function SET-UP is called. The terrain data is then read from a user-specified disk file using a user-specified input method. Figure 28a illustrates output.

b. Finding Ridges

Function SET-UP continues by calling function FIND-ALL-RIDGES, which tests all edges in the ground volume for their status as ridge lines. The test results are put in the data structure representing the edge. Figure 28b shows sample text output.

c. Making Air Volumes Convex

Function SET-UP continues by calling function MAKE-CONVEX-VOLUMES. This makes vertical planes through each ridge line, and then intersects all air volumes with this plane. This produces convex air volumes from non-convex air volumes as shown in Figure 29. This is the only time that non-convex intersection can occur, and the more restrictive intersection requirements for non-convex volumes are enabled here, as described in Chapter III, Section C1-c. Figure 28c shows sample output.

TABLE 7. GLOBAL VARIABLES

Variable Name	Contents
ORIGINAL-INPUT-VOLUMES	The names of input volumes before manipulations.
CONVEX-VOLUMES	The names of volumes after the air volumes were made convex.
FINAL-VISIBILITY-REGIONS	The names of all visibility volumes after visibility determination.

```
>>>> Volume created: |volume0002|   Composition: GROUND
>>>> Volume created: |volume0003|   Composition: AIR
```

(a)

```
Find all ridges in ground terrain:

Ridge check, line: |line0010|
Ridge check, line: |line0009|
Ridge check, line: |line0008| --  Ridge
Ridge check, line: |line0007|
```

(b)

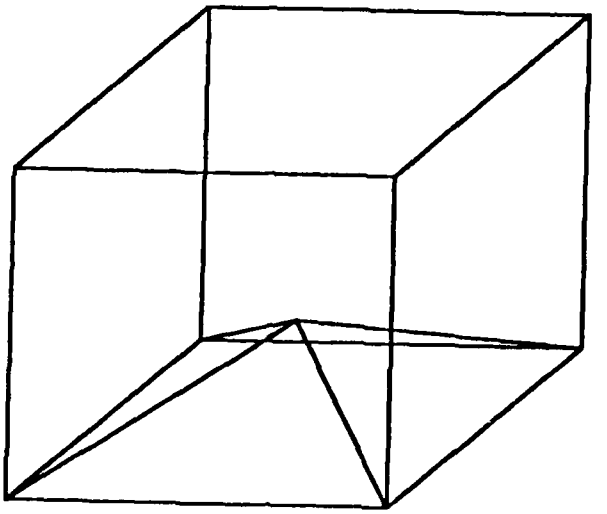
```
Making air volumes convex:

Air volumes: ((|volume0011|) (|volume0013|))
Ridge planes: (|plane0113| |plane0114| |plane0116| |plane0117|)

intersecting: (|volume0014| (1/2000 1/1000 0 1)) --- Result: (|facet0071|)
intersecting: (|volume0011| (1/2000 1/1000 0 1)) --- Result: nil (late 1)
intersecting: (|volume0011| (1 -2 0 0)) --- Result: (|facet0075|)
intersecting: (|volume0019| (1 -2 0 0)) --- Result: nil (late 1)
intersecting: (|volume0018| (1 -2 0 0)) --- Result: nil (late 1)
```

(c)

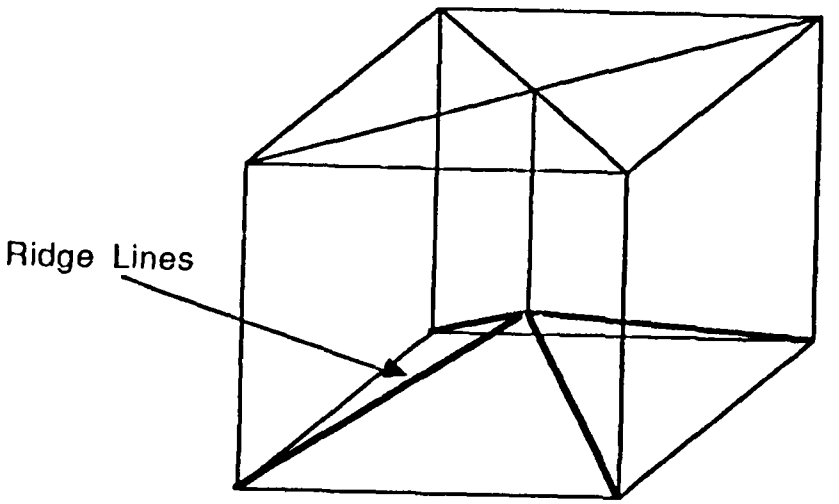
```
making visibility regions for: |observer0002|

Air volumes: ((|volume0005|) (|volume0004|))
Limiting planes of visibility: (|plane0018|)

intersecting: (|volume0005| (-1/750 0 1/300 1)) --- Result: (|facet0023|)
intersecting: (|volume0004| (-1/750 0 1/300 1)) --- Result: (|facet0028|)
```

(d)

Figure 28. Text Output



Non-Convex Air Volume
(a)



Convex Air Volume
(b)

Figure 29. Making Air Volumes Convex

d. Enter Observer Data

This is a manual user step to enter the location and effectiveness of observers to be used in the visibility model. This is done using function INIT-OBSERVER. Location is entered as an (x,y,z) coordinate, and observer effectiveness is entered as a decimal probability of detection between 0.0 (no probability of detection) and 1.0 (100% probability of detection). An observer flavor is instantiated by this function, and added to the universe instantiation.

e. Creating Visibility Volumes

A call to function SET-UP-2 will automate the rest of the static-visibility determination. This function will first call function MAKE-VISIBILITY-REGION for each observer entered by the user. This completes the visibility determination algorithm described in Chapter III. Figure 28d illustrates textual output from this function.

f. Finding Visibility

SET-UP-2 then calls function DETERMINE-VISIBILITY for each observer. This function determines the line-of-sight (LOS) from the center of each visibility volume to the observer. If the LOS is not blocked, the name of the observer is added to a list of observer names in the volume data structure. Figure 30a shows a sample output.

Function SET-UP-2 then calls function PROBABILITIES-ASSUMING-INDEPENDENCE-OR [Ref. 2], which

```
Visibility determination for: |observer0002|

  |volume0007| visible
  |volume0009| not visible
  |volume0008| visible
```

(a)

```
Determine Probability of Detection for visibility volumes

|volume0008| has P.D.: 0.75
|volume0009| has P.D.: 0.0
|volume0002| has P.D.: 0.0
```

(b)

```
Connecting volumes:

|volume0006| Connected to: (|volume0007| |volume0008|)
|volume0007| Connected to: (|volume0006|)
|volume0008| Connected to: (|volume0009| |volume0006|)
|volume0009| Connected to: (|volume0008|)
|volume0002| Connected to: NIL
```

(c)

Figure 30. Text Output

combines the probabilities of detection for all observers which can see a given visibility volume (using the information noted earlier in the data structure of the visibility volume). Figure 30b shows a sample output.

g. Connecting Volumes

Finally, SET-UP-2 calls function CONNECT-VOLUMES. This function builds the search graph by annotating the data structure of each visibility volume with the names of the visibility volumes to which it is adjacent, using the algorithm described in Chapter III, Section A2-c. figure 30c shows a sample output.

2. Path Planning

The path-planning phase is less structured than the visibility determination phase, so no overall control function has been implemented. The user must follow a minimum sequence of actions, as shown below.

a. Entering Start and Goal Points

These points are entered using function INIT-POINT.

b. Volume Path Planning

Function A-STAR-SEARCH performs A* search for the volume path through the visibility volumes. This function will instantiate a unique name for the path created, make the initial guess at the piecewise-linear path, and place that information into the paths data structure.

An alternate function, A-STAR-SEARCH-M, finds multiple volume paths from the start point to the goal point. The paths may be compared at a later time.

c. Optimizing the Path

Finally, the user may call function OPTIMIZE-PATH which will conduct one Snell's Law optimization pass along a given volume path. This optimization function will create a new instantiation of the path flavor for the optimized path. the user may call this function repeatedly to obtain the effect of multiple optimization passes. New paths are created on each call to let the user compare paths.

D. VOLUME INTERCEPT FUNCTIONS

The functions which subdivide a volume by a plane, while not directly called by the user, form the heart of the visibility determination algorithm. MAKE-CONVEX-VOLUMES and DETERMINE-VISIBILITY are the functions which directly call the series of volume intercept functions. These user-level intercept functions develop lists of volumes to be subdivided and lists of intercepting planes. The highest level intercept function, INTERSECT-ALL-PLANES-WITH-VOLUMES conducts repeated intercepts of the volumes with the intercepting planes. The resultant volumes replace the original volume in the list of volumes to be intercepted.

The volume intercepting algorithm described in Chapter III is implemented by the lowest level intercept function INTERSECT. This function takes one volume and one plane, and

performs their intercept. Text output from this function is of the format:

Intercepting: {volume-name} {A B C Ao} --- Result: {result-code}

where A, B, C and Ao are the constant coefficients of the intercepting plane. Figures 28c and 28d show examples of this output. Possible result codes are shown in Table 8.

Numeric errors can occur during the intercept function. They are the result of a rounding error associated with the use of floating point numbers in the calculation of point values and various coefficient values for vectors, edges and planes. This results in points which do not lie in planes that they should, lines that do not connect though they should, vectors that are equal being found not equal, etc. When a numeric error occurs, the intercept function will simply indicate that no intercept has occurred, a degenerate intercept condition is produced, and execution is not halted. The intercepting plane which caused the error is flagged for later use.

A method was developed to attempt to correct these numeric errors when they occur. This method adjusts the allowed imprecision of calculations, broadening the definition of equality between numbers [Ref. 18].

A=B if and only if (A - B)/B < {fixed constant} (4-1)

TABLE 8. INTERCEPT RESULT-CODES

Code	Description
nil (early1)	Invalid intercepting plane.
nil (early2)	Intercept along a facet of a convex volume.
nil (late 1)	Intercept at a point (only).
nil (late 2)	Not enough facets in resultant volume. Usually indicates that a numeric error occurred.
({facet-name})	Successful intercept, facet named is the common facet between the resultant volumes.
({facet-name-1} {facet-name-2}...)	Successful intercept, except that, due to a numeric error, several facets are common to the resultant volumes.
Violated Euler's Relation #.##	A numeric error occurred at precision #.##.

The value of the fixed constant value is increased incrementally to fix a numeric error. The normal fixed constant is initially 0.25%. While correcting an error this will be increased incrementally up to a maximum of 1.0%.

If the correction method is unsuccessful in correcting the numeric error, then no further action is taken. This can have a significant impact on the validity of the visibility model, and is discussed further in Chapter VI.

V. RESULTS

Two useful metrics were developed to measure the effectiveness of the program. The number of visibility volumes built is compared to the number of visibility cubes created in a uniform-cube approach to the visibility problem to measure the effectiveness of the visibility-determination phase. The weighted average probability-of-detection for the path is compared to the maximum probability-of-detection for the path to measure the effectiveness of the path planning.

A. VISIBILITY VOLUMES

Six different airspace models were tested in the visibility-determination phase, each in three different situations, for a total of 18 test cases. Twelve of these test cases cover airspace over simple terrain models with isolated, idealized terrain features. These include various simple ridge and peak shapes. The remaining test cases represent more complex terrain. All terrain was bounded inside a 1000 by 1000 by 1000 cube (of no units).

Airspace models were tested after air volumes were made convex, after visibility processing for one observer, and after processing for two observers. The number of visibility volumes was recorded after each case. The effectiveness of the visibility model was judged against the reduction in size of the search graph, relative to a standard maximal search

graph. This standard maximal search graph is composed of 10 by 10 by 10 cubes (800,000 nodes), or 25 by 25 by 25 cubes (52,100 nodes), or 50 by 50 by 50 cubes (6,400 nodes). These assume an initial air volume from ground level at 200 units to a maximum altitude of 1000 units. The presence of terrain features represents a 25% decrease in the size of the air volume, so the final sizes of the standard maximal search graphs are 600,000, 38,400 and 4,800 nodes, respectively.

Table 9 shows the results obtained for the airspace models. Figure 31 shows one of these models (a simple ridge, called a "single ridge" in Table 9) after the air volumes have been made convex. Figure 32 illustrates the same model after processing for a single observer (the location of the observer is noted with a lozenged OBS symbol). Figure 33 and 34 show the same views with another terrain model (a complex ridge structure called a "full ridge" in Table 9) with two observers. It should be clear from Figure 34 that a graphical display of the visibility regions in even a simple problem is too confusing to be useful.

Table 9 indicates that the number of visibility volumes grows explosively as a function of the number of ridge lines and the number of observers. The graphs of Figure 35 clearly indicate a strong functional relationship between these quantities. The growth in the numbers of visibility volumes approximates an exponential curve, as a function of the number of ridge lines, as shown in Figure 35b. Such growth

TABLE 9. VISIBILITY VOLUMES BY TERRAIN TYPE

Terrain Type/ Number of Ridge-lines	Number of Convex Volumes	Number of Visibility Volumes (One Observer)	(Two Observers)
Single Ridge/1	2	4	6
Double Ridge/2	3	7	9
Peak/4	4	24	94*
Full Ridge/5	10	70	177*
Box Canyon/5	13	76*	198*
Double Peak/8	10	109*	502*

Note: * marks terrain models with numeric errors (defined in Chapter Four, section D)

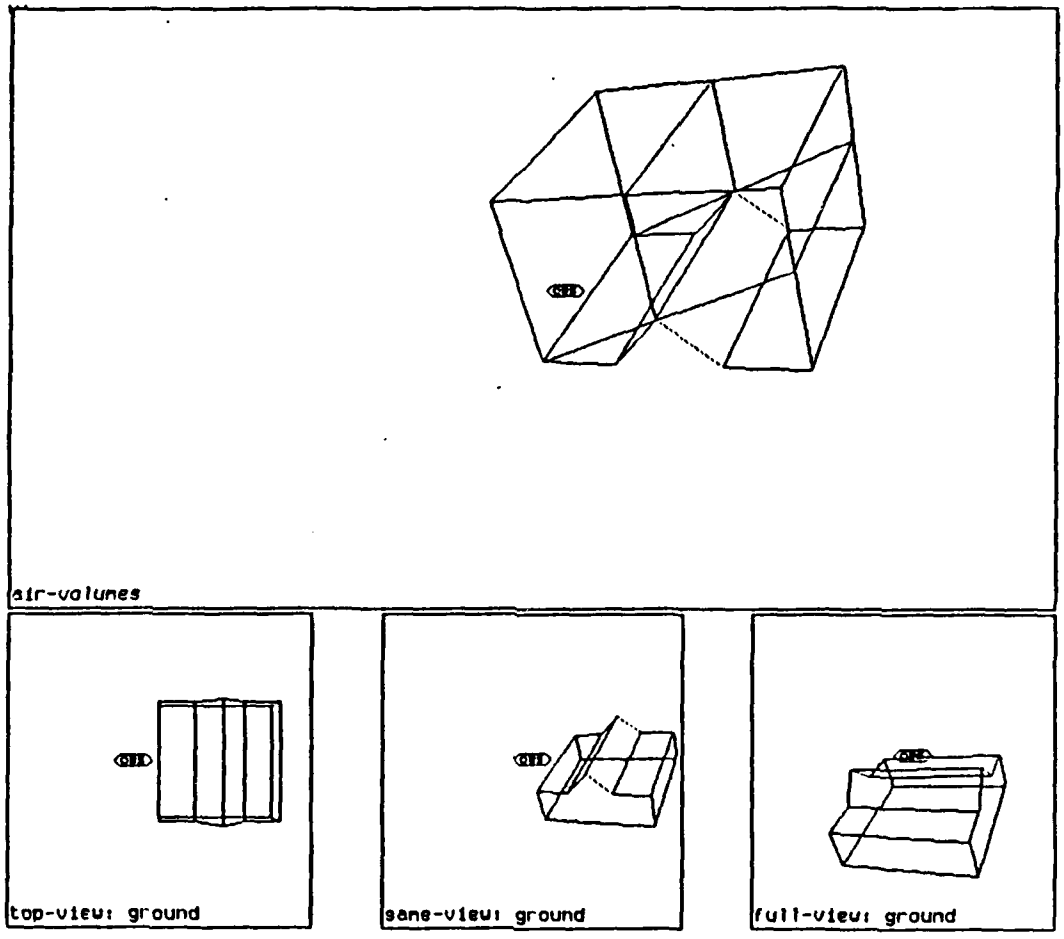


Figure 32. Visibility Volumes with One Observer over a Single Ridge

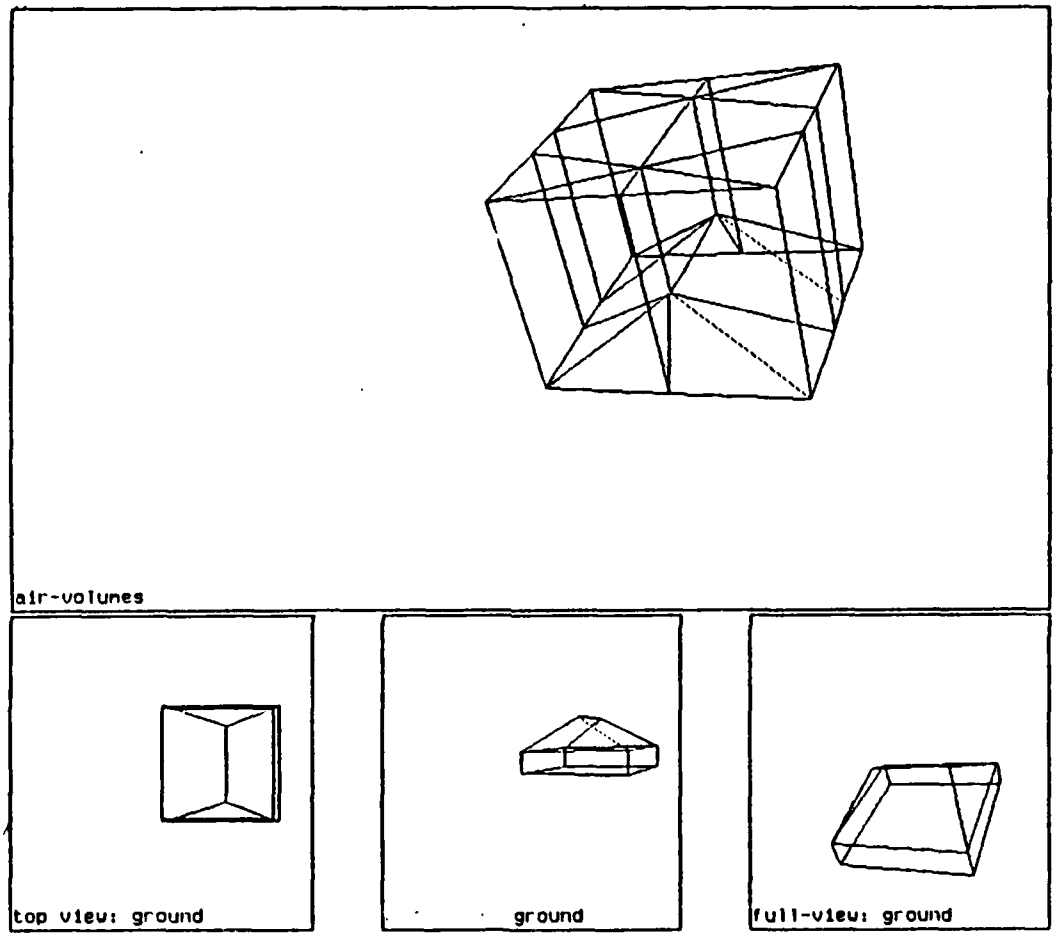


Figure 33. Convex Air Volumes over a Full Ridge

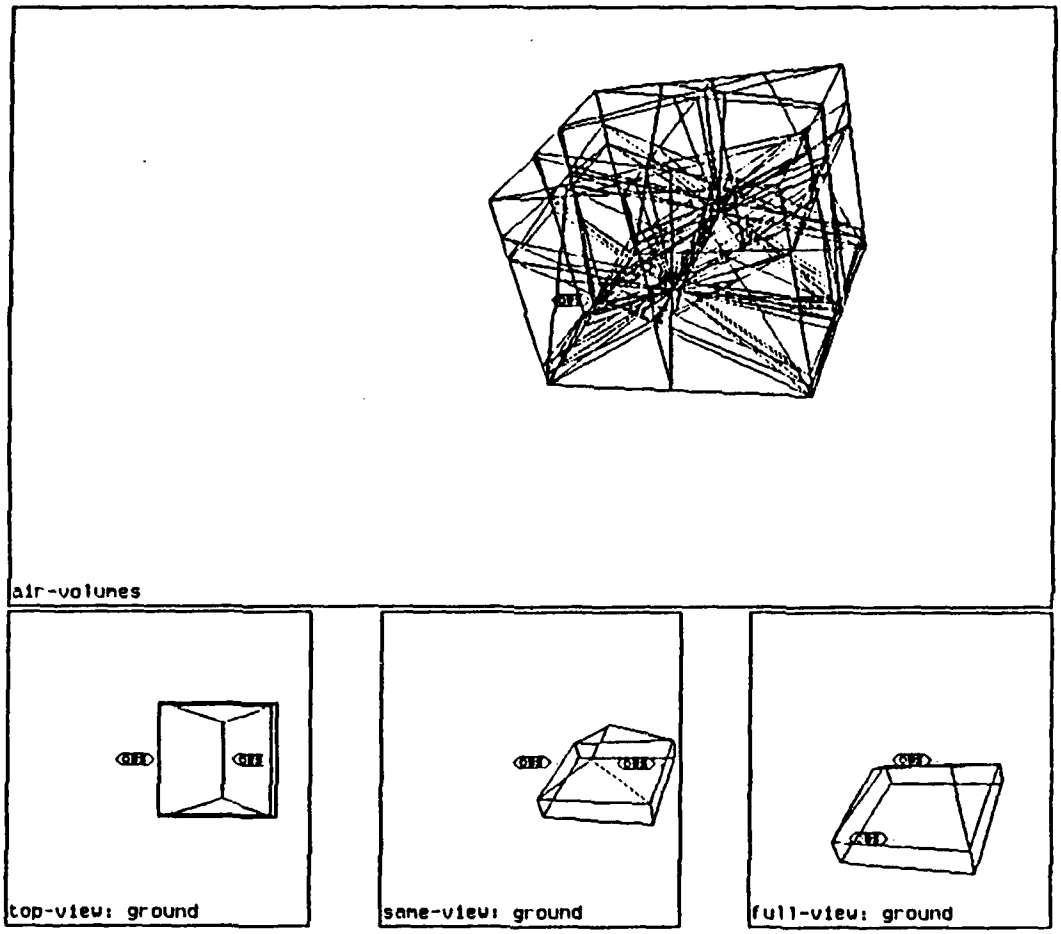
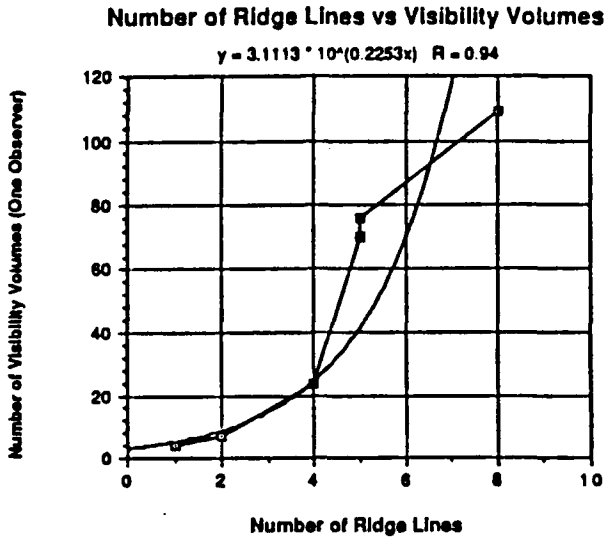
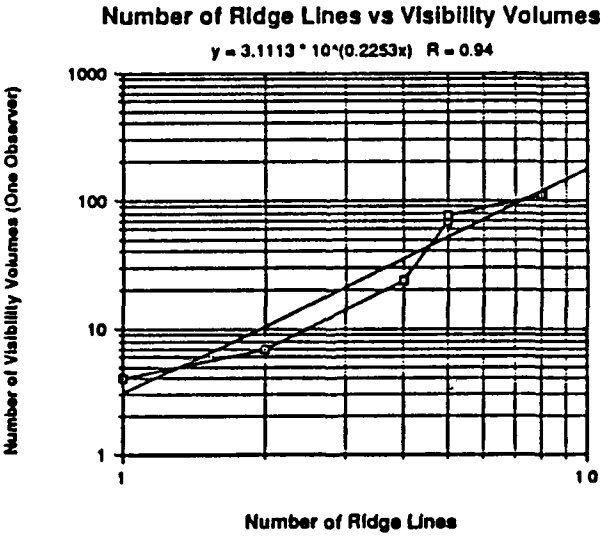


Figure 34. Visibility Volumes for a Full Ridge with Two Observers



(a)



(b)

Figure 35. Plots of Ridge Lines vs Visibility Volumes

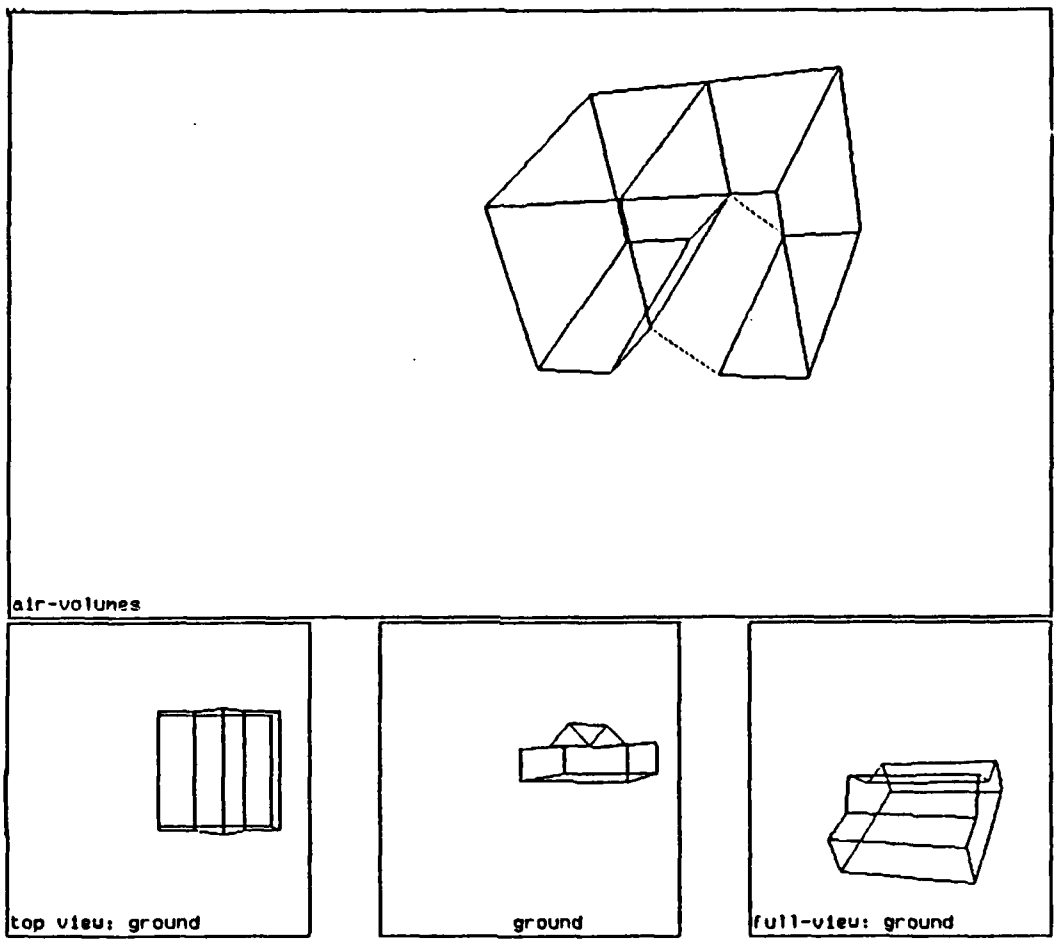


Figure 31. Convex Air Volumes over a Single Ridge

is relatively innocuous with simple terrain models, but with more complex models this growth hurts the overall performance of the path-planning program. This is discussed in more detail in the following chapter.

Note the increased incidence of numeric errors (as defined in Chapter III, Section D) encountered as the number of ridges and observers increases, as shown in Table 9. These errors present potential deviations from the ideal visibility model. Volumes affected by numeric errors may not have uniform visibility for all enclosed points.

The visibility model does significantly reduce the size of the search graph from the standard maximal graph, as indicated in Table 10.

B. PATH PLANNING

Representative results for the path planning phase are shown in Table 11 and Table 12. Figures 36 through 47 illustrate some of these paths. Three views are shown for each terrain model, one for the initial, non-optimal piecewise-linear path (an initial guess, based on the volume path, and defined in Chapter III, Section B1-c), one for the optimal linear path (defined in Chapter III, Section B1-c), and one showing both paths together.

Figure 48 shows the effect of varying the location of the start point while keeping the end point fixed. This illustrates the trade-off made during path planning between

TABLE 10. REDUCTION IN SEARCH GRAPH SIZE

Terrain	Relative size of search graphs (visibility model/standard model) with one Observer		
	10 unit cube	25 unit cube	50 unit cube
One Ridge	0.0	.0002	.0005
Two Ridge	0.0	.0002	.0015
Peak	0.0	.0007	.0050
Full Ridge	0.0	.0019	.0140
Box Canyon	.0002	.0026	.0203
Double Peak	.0002	.0024	.0228

TABLE 11. PATH PLANNING RESULTS FOR SIMPLE TERRAIN

Path	Terrain	Observers	Length	Probability of Detection Weighted Average	Maximum
Initial	Single ridge	1 (75% eff)	1764	0.47	0.75
Final optimize			1563	0.45	0.75
Initial	Double ridge	1 (75% eff)	2138	0.22	0.75
Final optimize			1719	0.21	0.75
Initial	Double ridge	1 (75% eff)	1630	0.75	0.75
Final optimize			1425	0.75	0.75
Initial & Final	Double ridge	1 (75% eff)	1922	0.35	0.75
Initial	Peak	1 (75% eff)	1932	0.0	0.0
1st optimize			1754	0.0	0.0
2ond optimize			1711	0.0	0.0
3rd optimize			1655	0.0	0.0
4th optimize			1630	0.0	0.0
5th optimize			1615	0.0	0.0
Final optimize			1605	0.0	0.0

TABLE 12. PATH PLANNING RESULTS FOR MORE COMPLEX TERRAIN

Path	Terrain	Observers	Length	Probability of Detection	
				Weighted Average	Maximum
Initial	Box Canyon	1 (75% eff)	2778	0.12	0.75
Final optimize			2548	0.02	0.75
Initial	Box Canyon	1 (75% eff)	709	0.49	0.75
Final optimize			615	0.38	0.75
Initial	Box Canyon	1 (75% eff)	2128	0.12	0.75
First optimize			1943	0.32	0.75
Final optimize			1910	0.02	0.75
Initial & Final	Double Peak	1 (75% eff)	2342	0.29	0.75

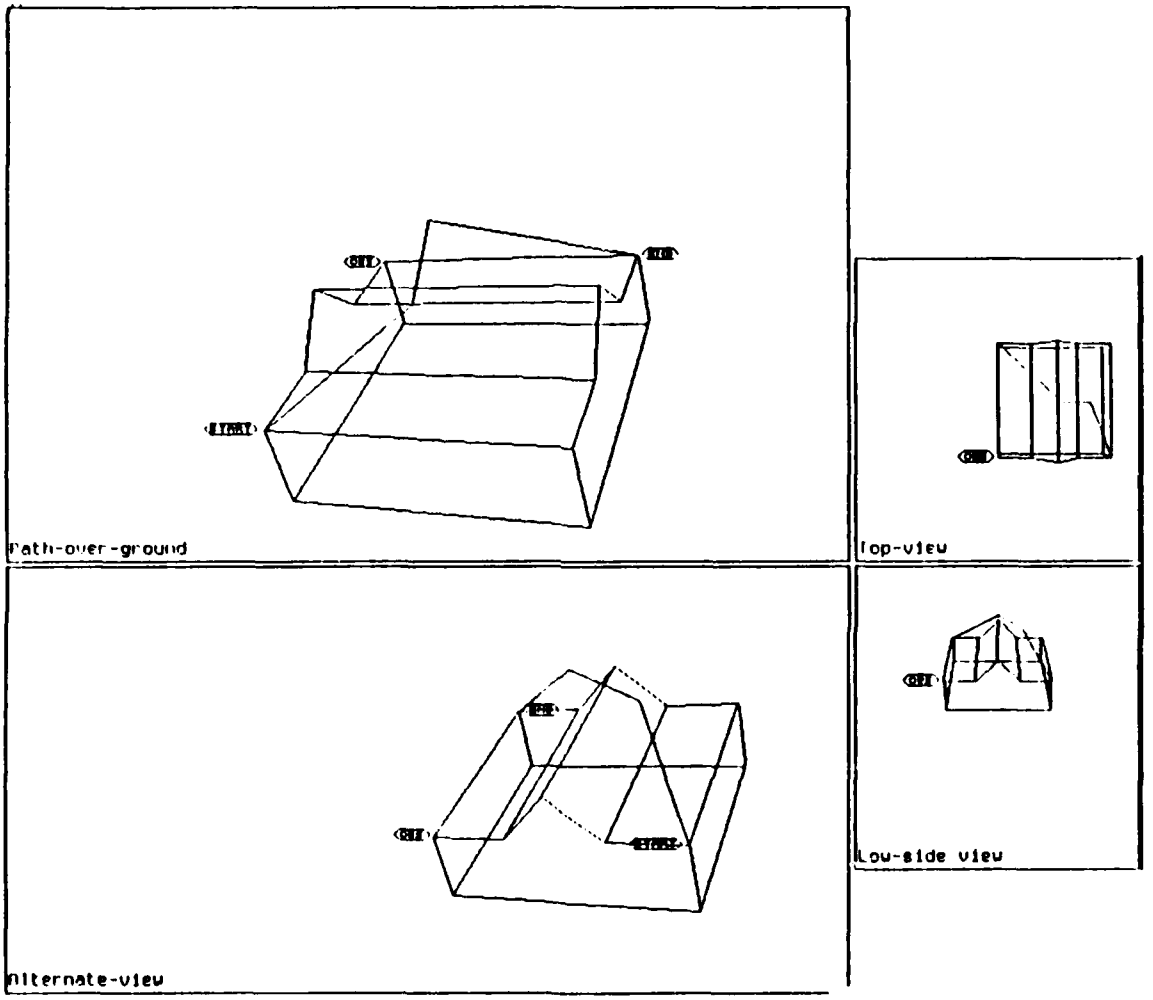


Figure 36. Initial Piecewise-Linear Path over a Single Ridge

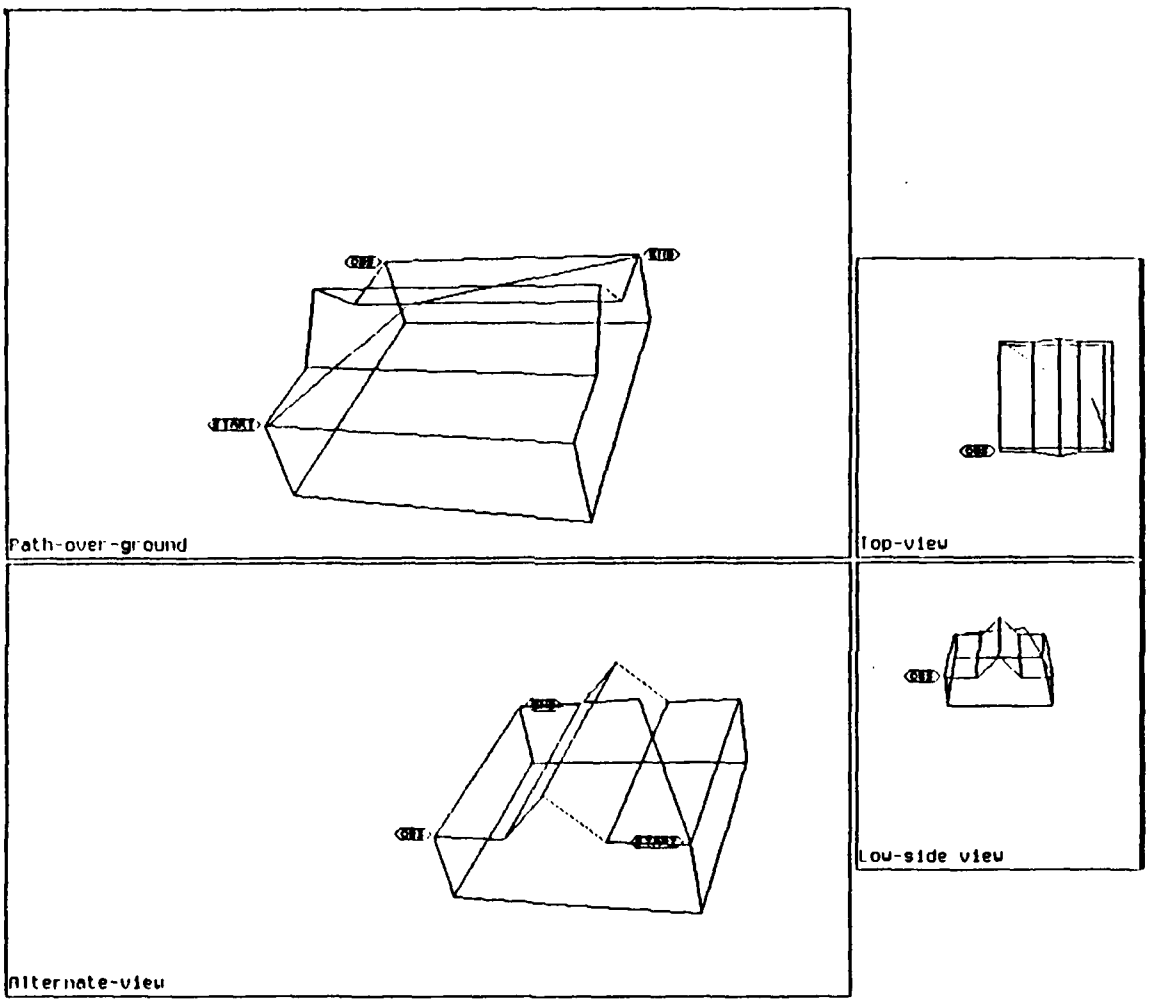


Figure 37. Final Piecewise-Linear Path over a Single Ridge

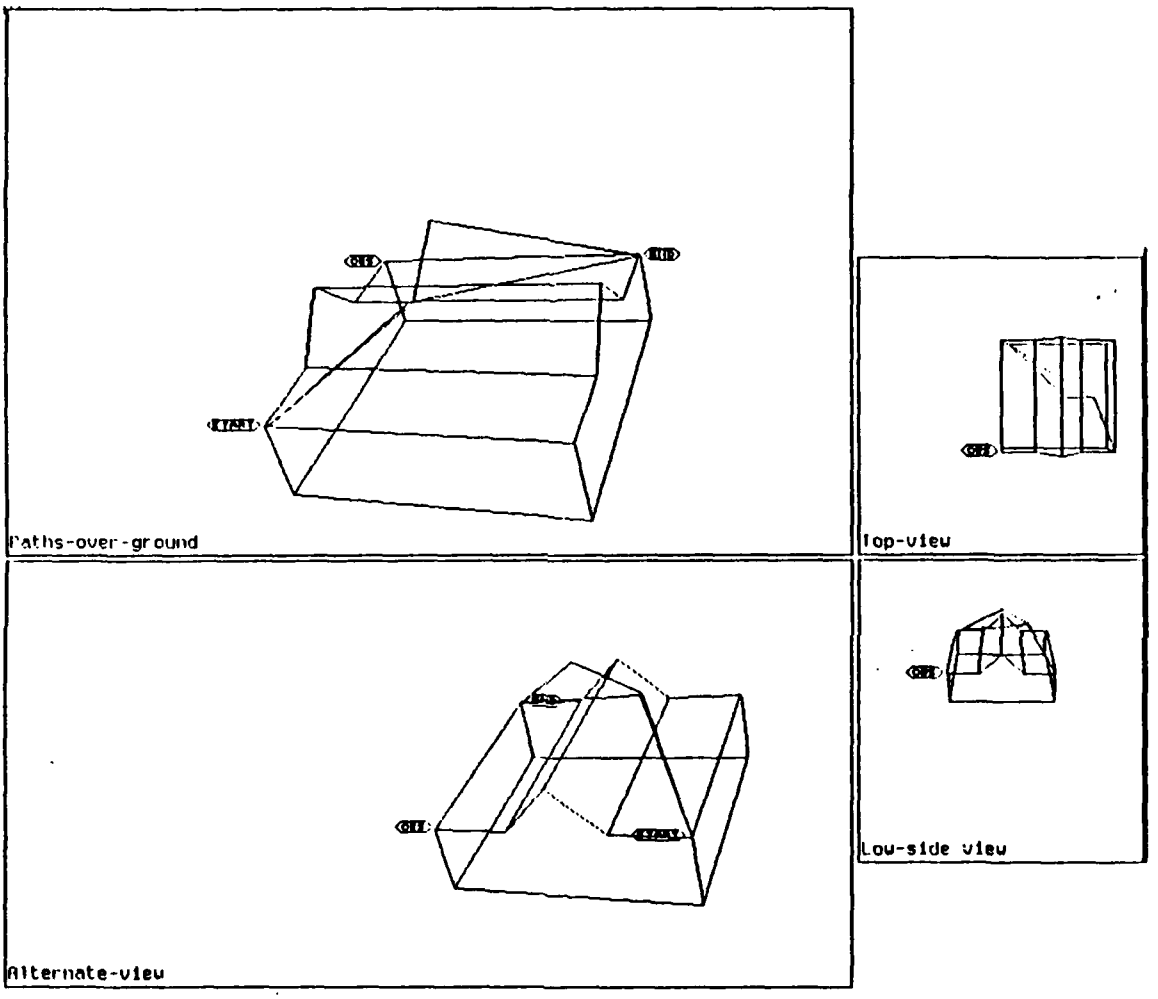


Figure 38. Combined View of Paths from Figures 36 and 37

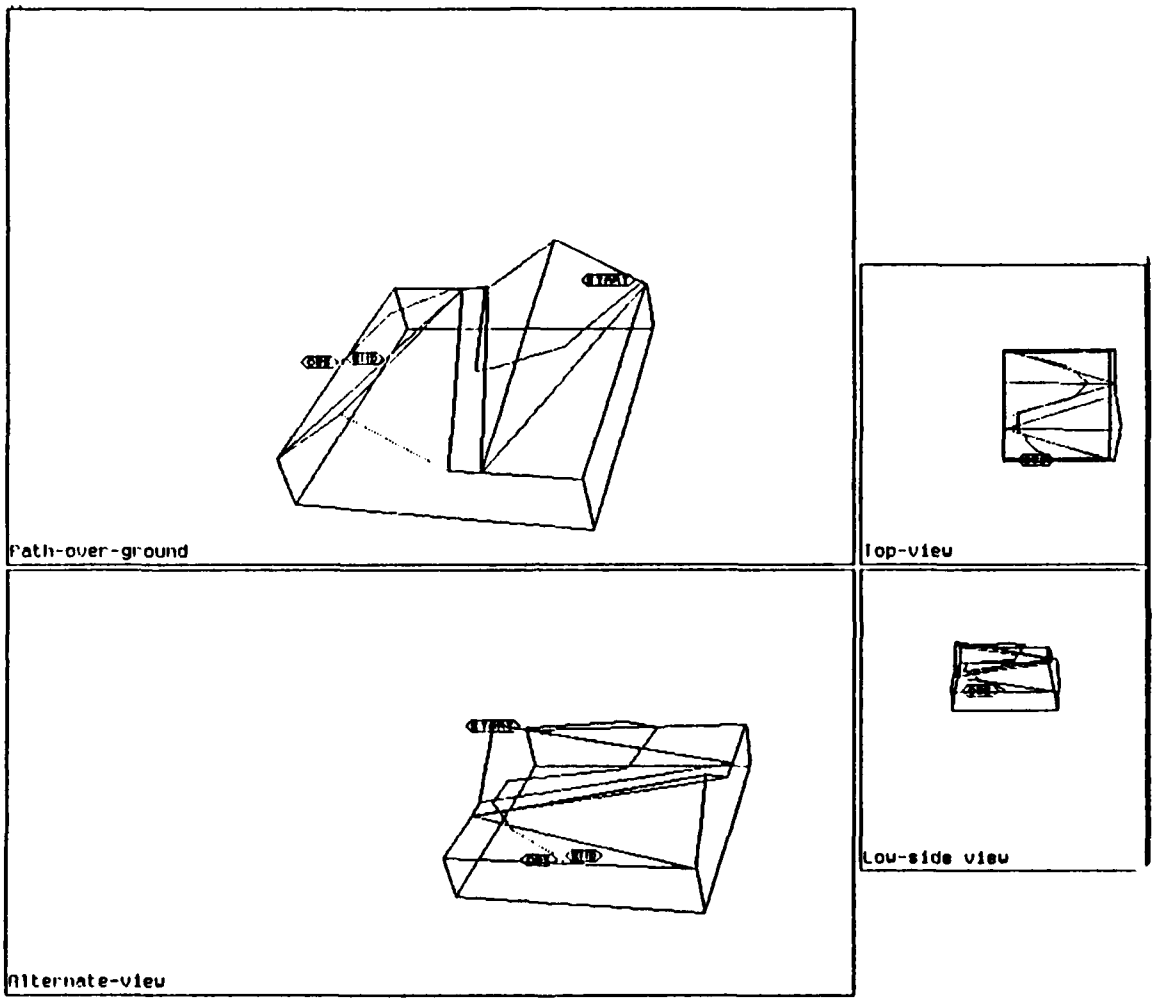


Figure 39. Initial Piecewise-Linear Path over a Double Ridge

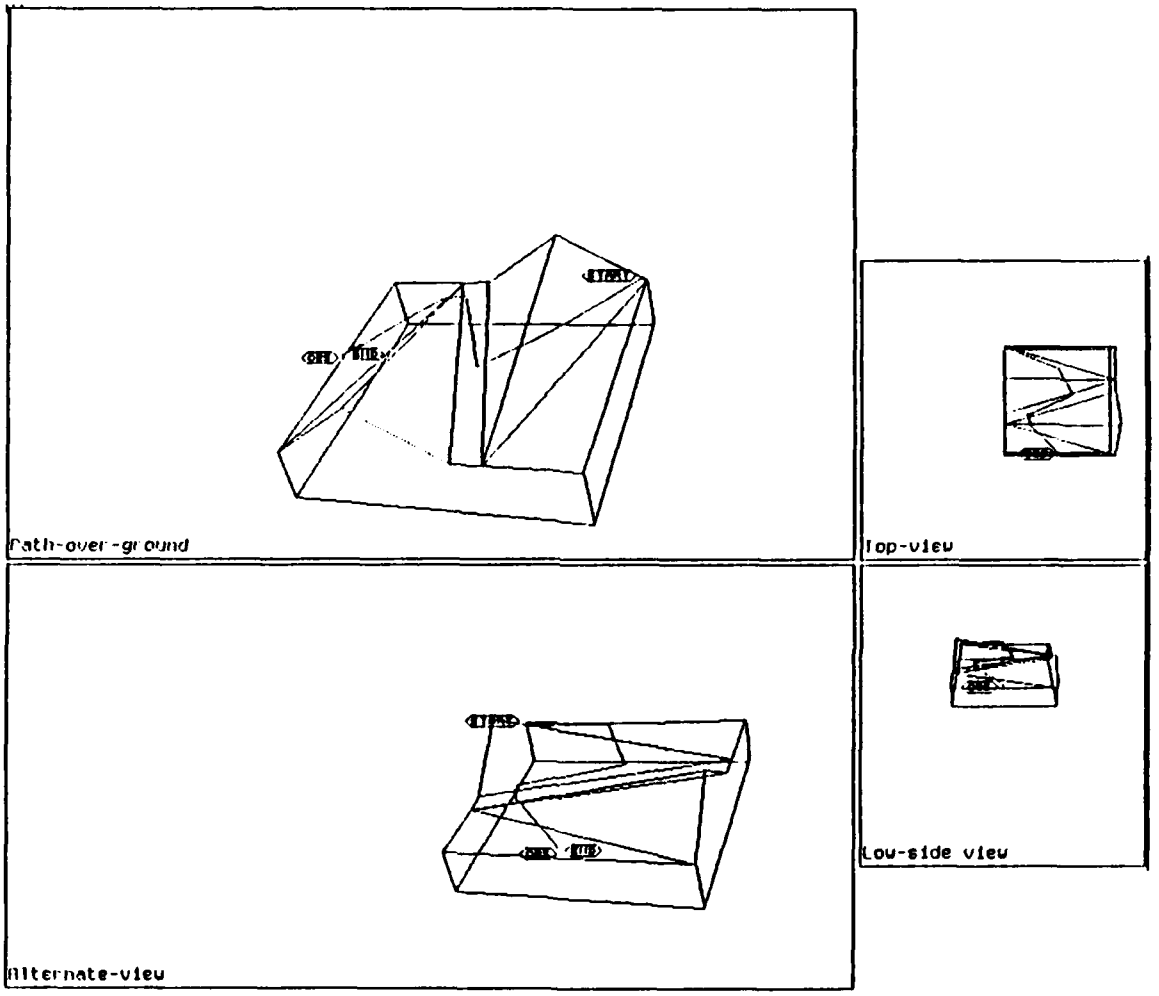


Figure 40. Final Piecewise-Linear Path for a Double Ridge

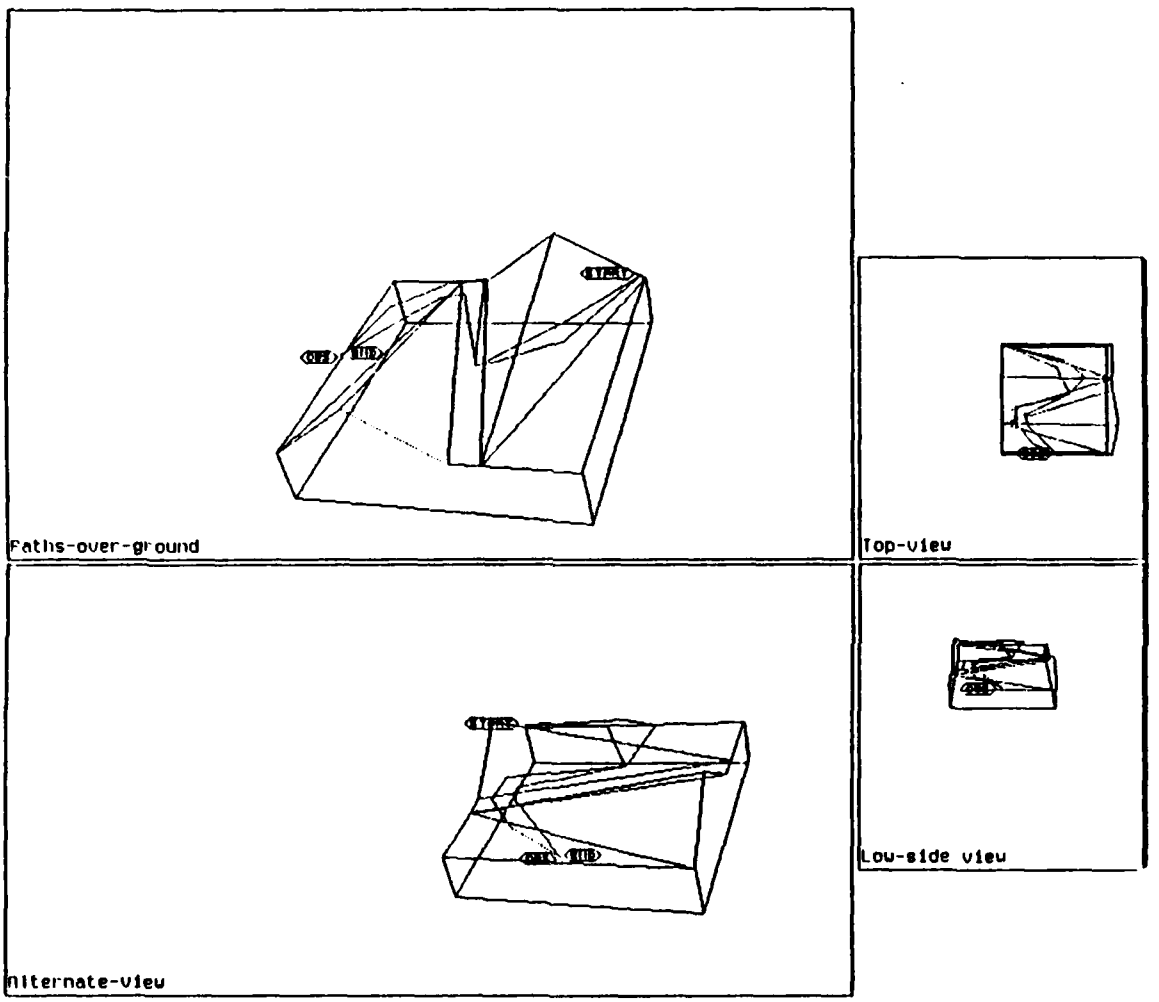


Figure 41. Combined View of Paths for Figures 39 and 40

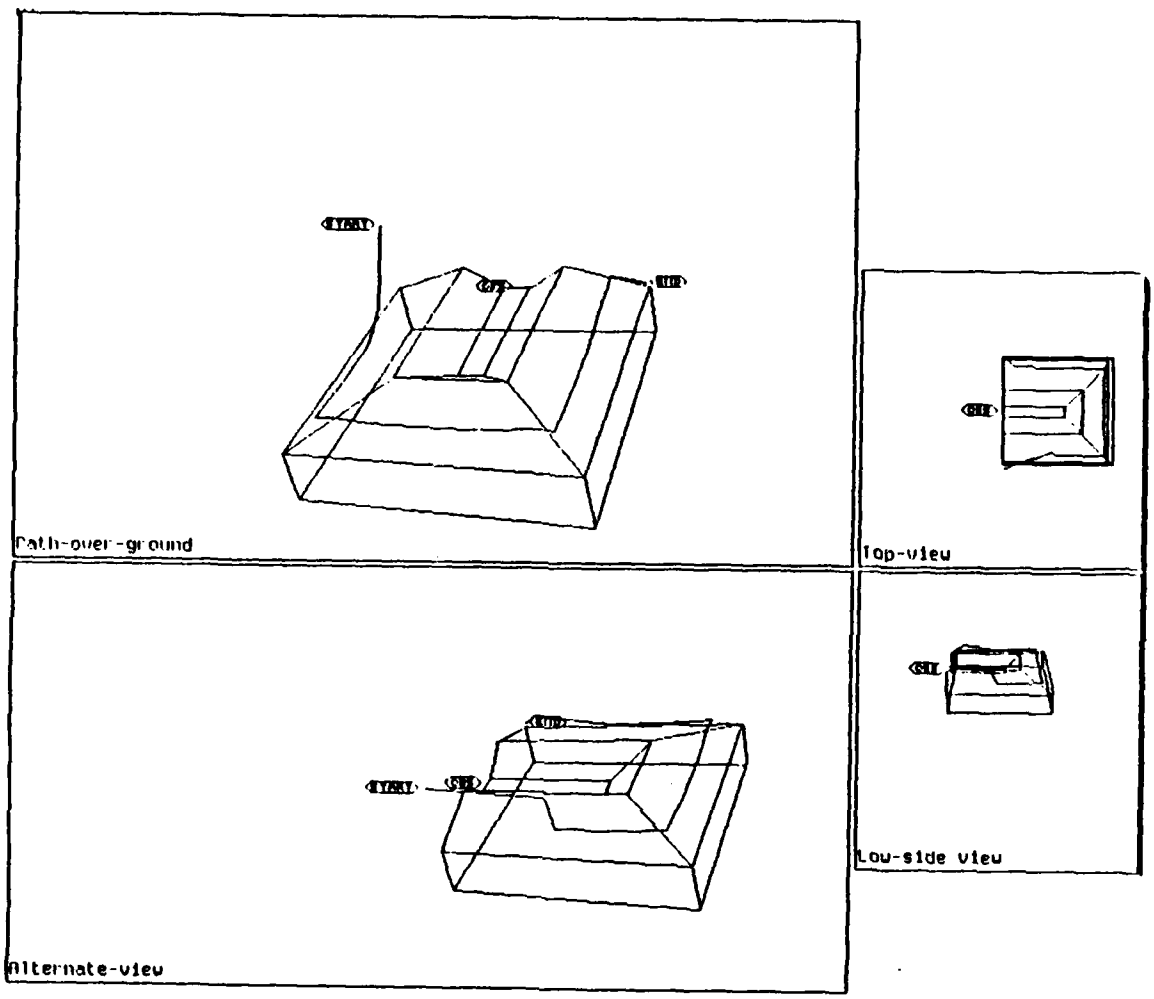


Figure 42. Initial Piecewise-Linear Path over a Box Canyon

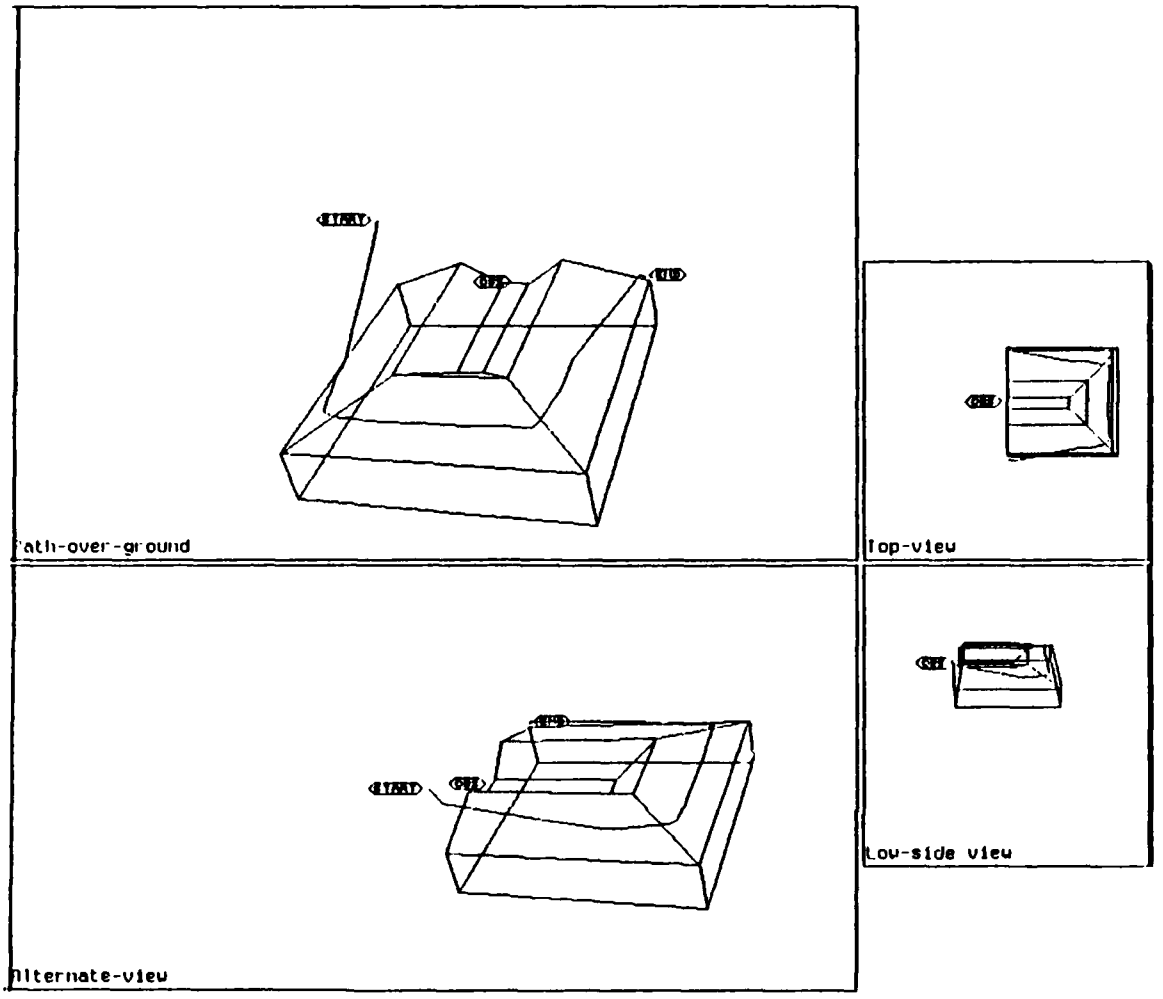


Figure 43. Final Piecewise-Linear Path over a Box Canyon

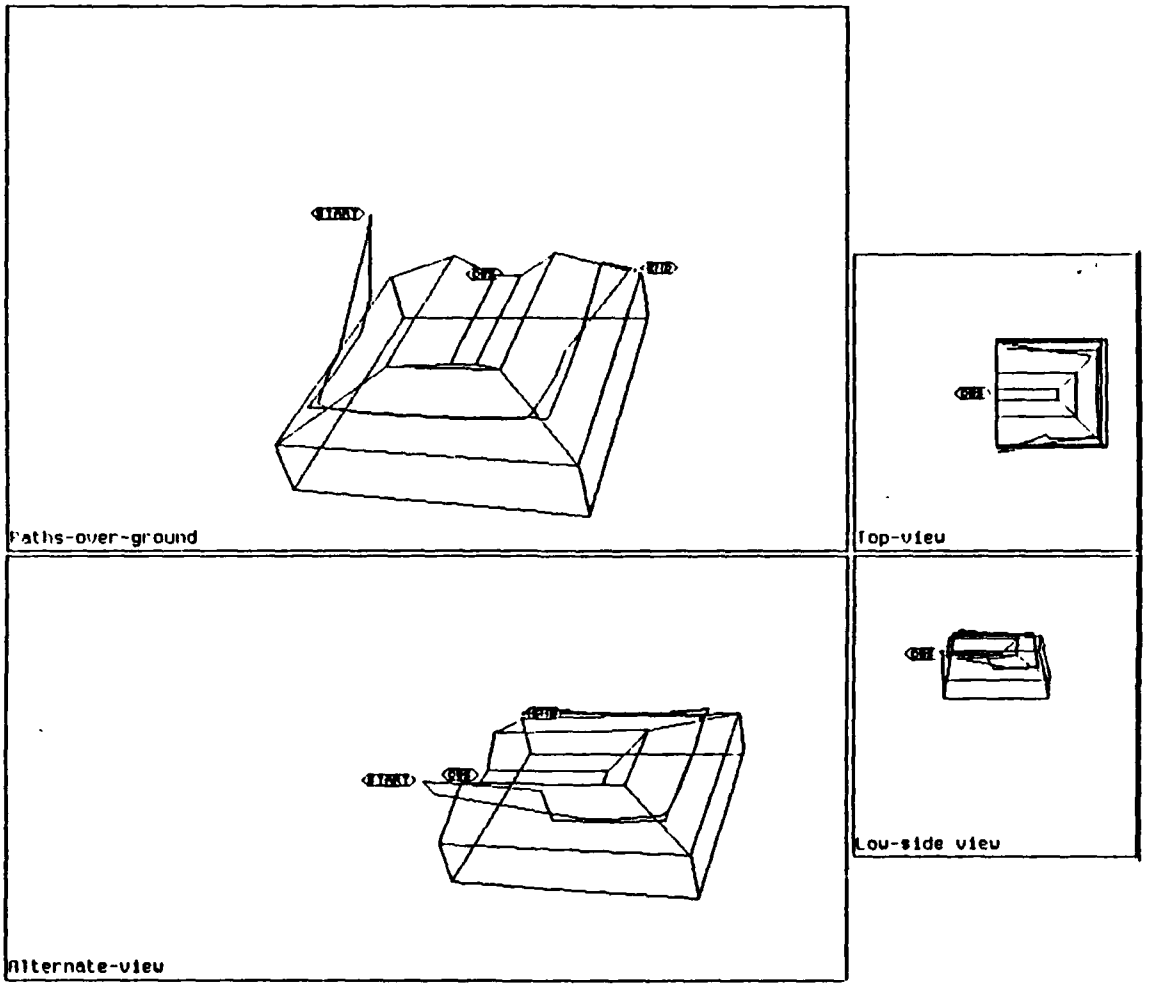


Figure 44. Combined View of Paths from Figures 42 and 43

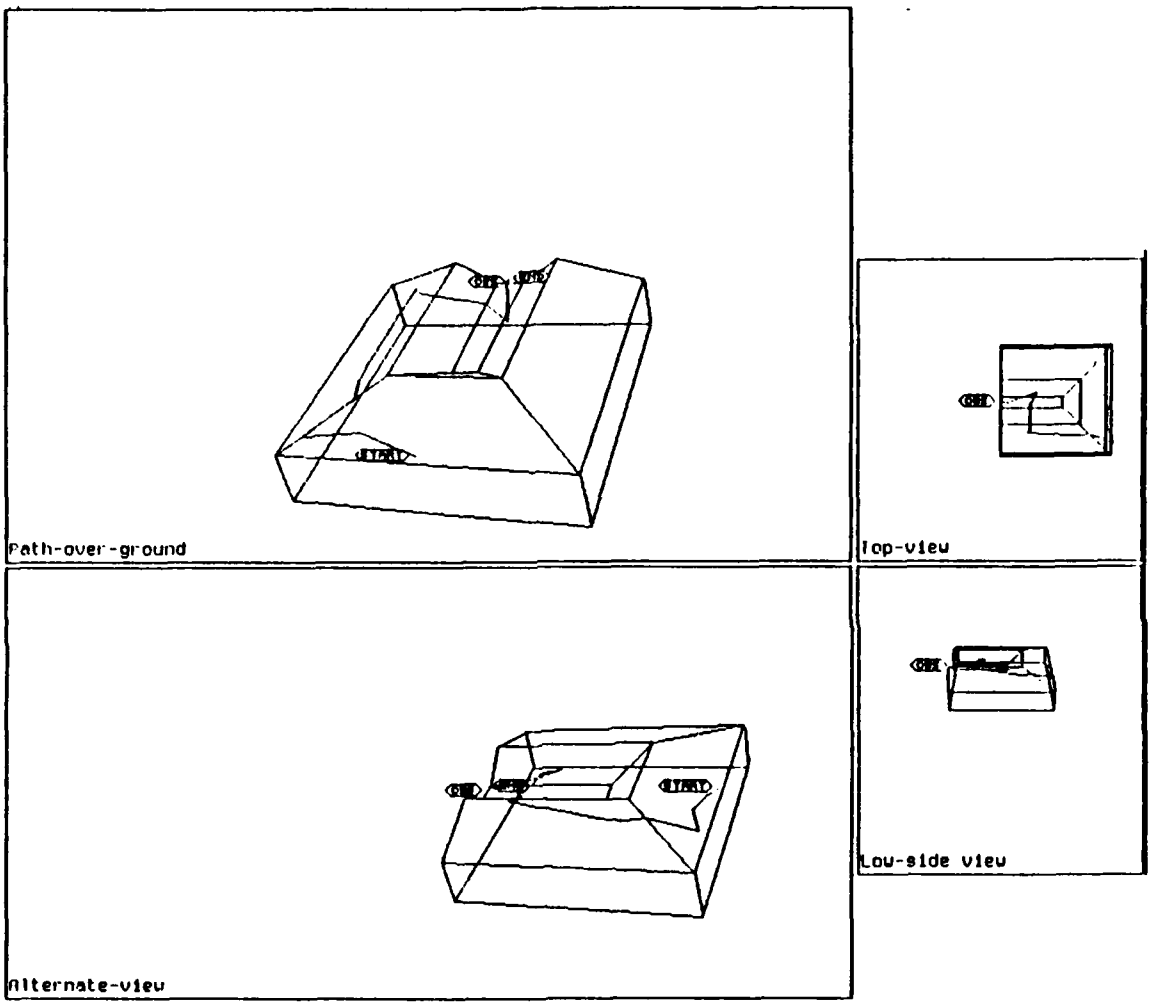


Figure 45. Initial Piecewise-Linear Path over a Box Canyon

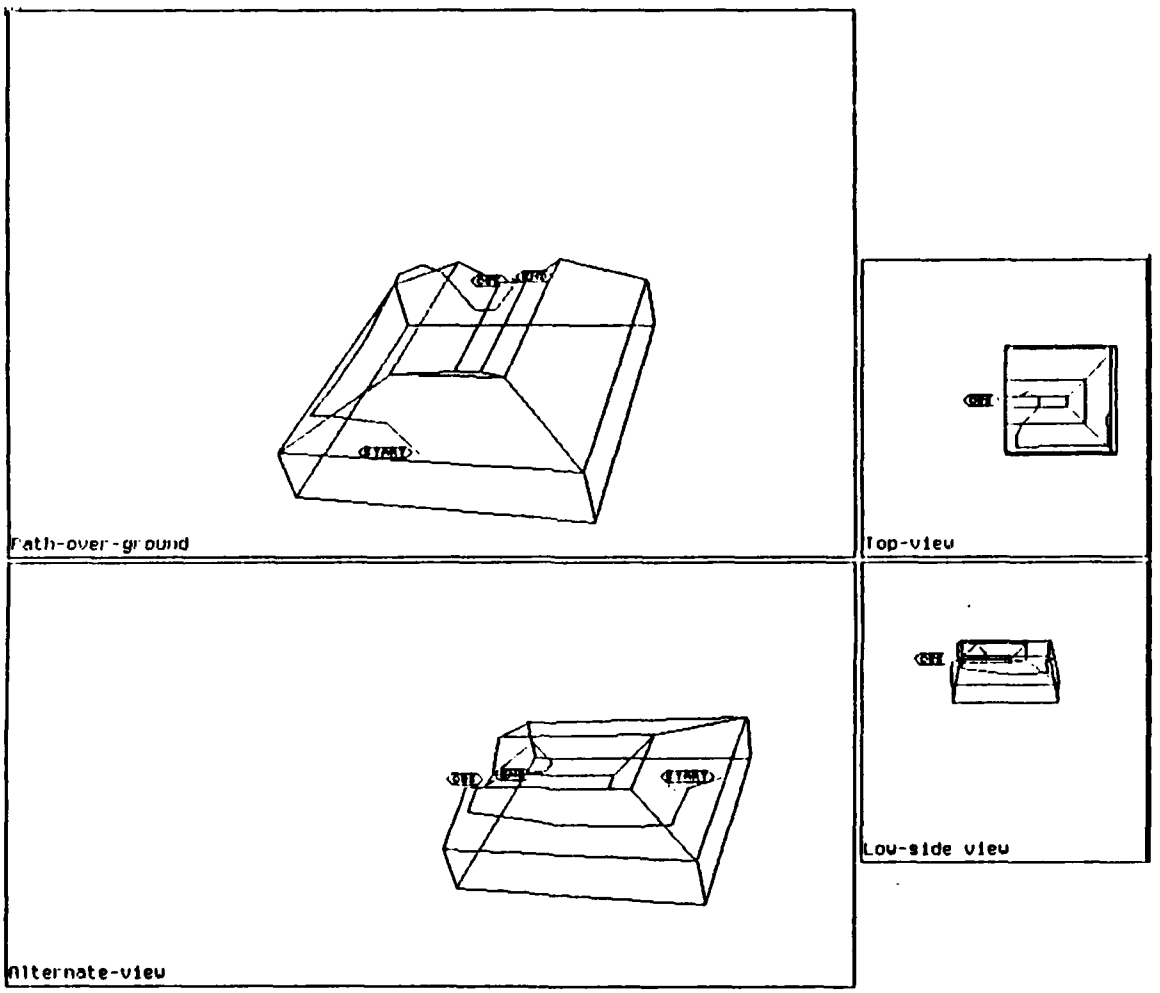


Figure 46. Final Piecewise-Linear Path over a Box Canyon

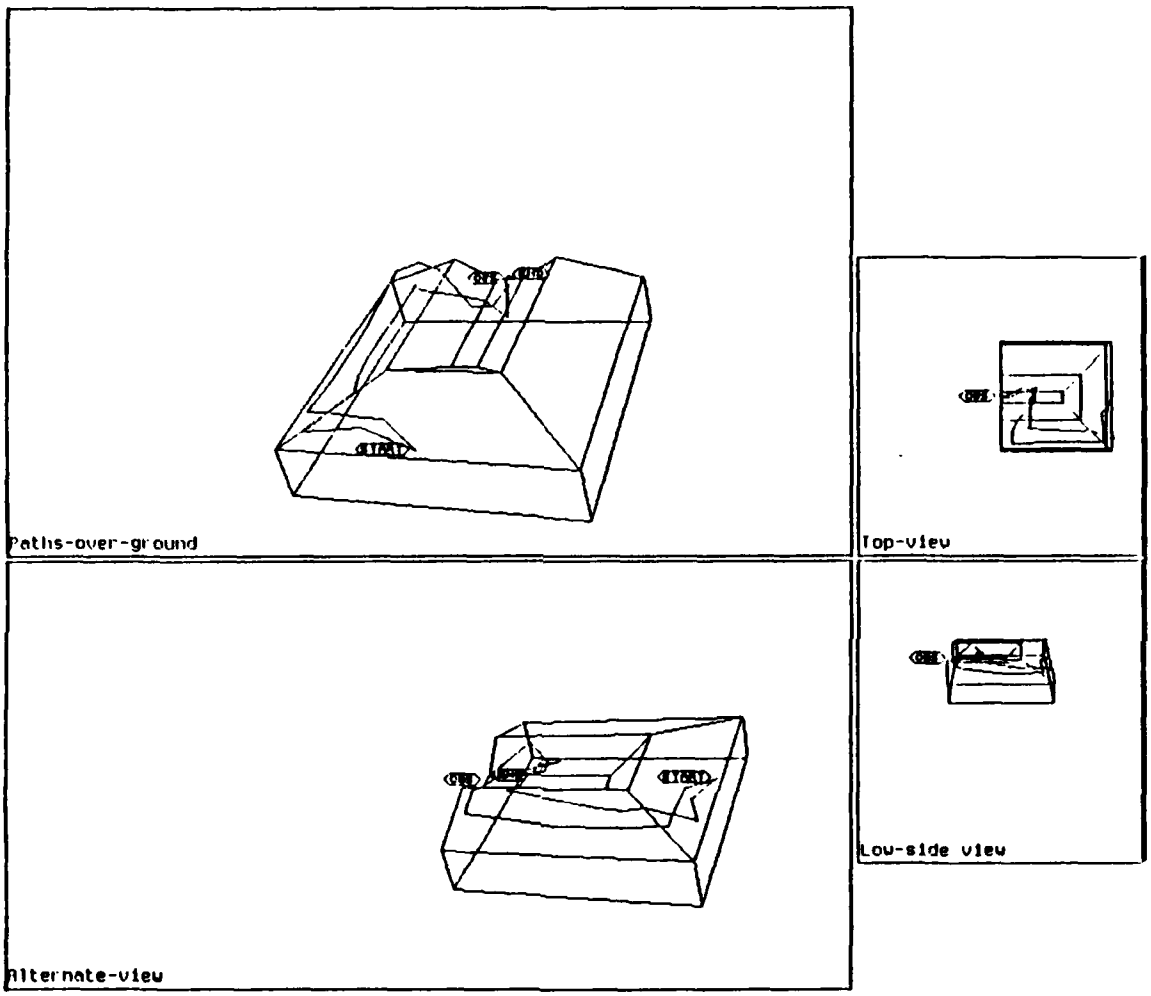


Figure 47. Combined View of Paths for Figures 45 and 46

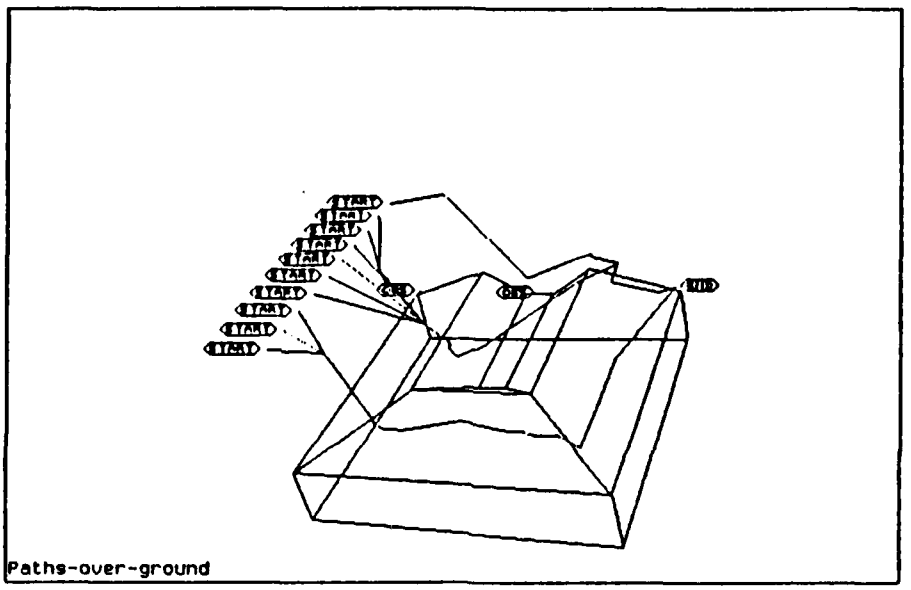


Figure 48. Paths Found with Different Start Points over a Box Canyon

distance travelled and probability of detection. All paths displayed are non-optimal initial guesses.

Tables 11 and 12 also show the effects of the Snell's Law optimization (of Chapter III, Section B1-c) on the cost and average probability-of-detection of the paths. On average, a 10% reduction in total cost is obtained as a result of optimization, with most of that gain being realized in the first optimization step. Figures 36 through 47 also show that the optimized paths generally remain in the same planes as the initial path. This is an artifact of the optimization procedure. Lower-cost optimal paths may result if this restriction is removed.

VI. CONCLUSIONS

It is clear from the previous chapter that the visibility-volume approach presented in this thesis can be successfully applied to the three-dimensional path planning problem. Reasonable paths were produced over a variety of terrain types. Search graphs were reduced in size compared to the alternate approach proposed in Chapter V (Section A), usually by two to five orders of magnitude. The search algorithm used was an easily implemented one, and no exotic pruning criteria were necessary during the path-planning process.

A. KNOWN PROBLEMS

Several shortcomings of this approach are worth noting, however. The growth in the number of visibility volumes is exponential as the number of ridge lines increases. This slows both the static-visibility determination and the path planning. As the number of visibility volumes increases, their average size decreases, relative to the size of the original air volumes. Small visibility volumes create narrow volume paths, and this decreases the effectiveness of the optimizing function by decreasing the area available to move the maneuver points.

The presence of very small visibility volumes reduces the chances that the A* search agenda will ever contain areas of

low probability of detection. If A* search never places a low probability-of-detection volume on the search agenda, it will not be able to evaluate the effectiveness of planning a path through that volume.

Occurrences of the numeric errors discussed earlier result in volumes containing regions of non-uniform visibility, which will result in paths being planned to pass through areas of high visibility in preference to areas of low visibility. Again, the extent of this problem was not examined in this thesis, but measures recommended to correct other shortcomings will help to alleviate this problem.

No true aerodynamic effects have been included in the energy model proposed in this thesis. Such models exist, but they were intentionally left out to avoid over-complicating the problem. The energy model used here predicts energy use in the volume path-planning stage, and only minimizes distance flown and visibility during the path optimization functions. Optimization must be extended to cover aerodynamic effects (turns, climb and dive effects). Note that this will require extensions to Snell's Law to account for these non-optical effects.

Not until an optimal piecewise-linear path is found is the precise nature of the flight path known. As a result, the best overall path may be through a non-optimal volume path (e.g., the second or third solution found by the A* search). The nature of this effect should be investigated.

Non-point observers, such as a small group of radars or dispersed groups of men, are not well handled by the visibility model. Since the model is very sensitive to an increase in the number of observers (which cause a rapid rise in the number of visibility volumes, and a corresponding decrease in the size of the individual volumes within the fixed airspace), simulating a large group of observers will be impossible to handle.

No diffraction, refraction, interference or diffusion effects are implemented. These can be significant effects, especially for radar and sonar observation. Simple range models can be implemented to account for some effects, while empirically-based changes to the probability-of-detection for certain visibility volumes can account for others. Diffraction effects due to terrain could be approximated by intersecting volumes with curved visibility-boundary surfaces instead of planes. Similar approximations can be made to correct lines-of-sight for various atmospheric effects.

Finally, the program is not fast. Construction of visibility volumes can take up to 24 hours to complete for complex terrain models (for the double peak terrain of Table 9 in Chapter V, consisting of eight ridge lines and two observers). Five to ten hours is the average run time for a simple terrain model (the peak in Table 9 in Chapter V, consisting of four ridges and one observer). Search times are very short, and run from ten seconds to several minutes,

on the average; large numbers of visibility volumes will take up to 15 minutes.

B. RECOMMENDATIONS

Corrective measures not already discussed to the problems mentioned above fall into several areas: better selection criteria for planes of visibility, a method of combining visibility volumes, parallel processing for some parts of the program, and modification of the search methodology used.

Algorithm 3-2 of Chapter III for the selection and use of limiting planes of visibility was not implemented in this thesis. It alone will not prevent rapid growth in the number of visibility volumes, but will serve to slow it some. That, combined with other measures (range effects, independent processing of terrain segments, etc.) may significantly control the number of visibility volumes.

A second corrective measure, coalescing visibility volumes after their creation will help resolve the small-visibility-volume problem. This would involve adjacent volumes with the same probability-of-detection. This could create non-convex visibility volumes, however, and must be applied with caution.

Much of the visibility-determination portion of the program involves intercepts of different volumes with the same intercepting plane. These calculations are functionally independent from one another, and thus are amenable to a parallel processing. Since several hundred such

interceptions occur, significant improvements in execution times could be expected with the use of several parallel processors. Parallel processing may also help to speed up the search portion of the program.

The two-phase path-planning methodology used here should be compared with a single-phase path-planning algorithm. Finding the optimal piecewise-linear paths as the A* search progresses may result in finding a better flight path.

C. DISCUSSION

There are two advantages to the volume-oriented approach to the three-dimensional path-planning problem: the abstraction of airspace to a search graph based on large, irregular visibility volumes, and the resultant simplification of the search.

Appropriate use of the visibility volume concept ensures that A* search can effectively evaluate all possible paths from the start point to the goal without missing large areas of the search graph. As mentioned earlier, A* cannot evaluate search nodes that do not get onto the search agenda, and the use of large, irregularly shaped visibility volumes ensures that the largest possible amount of airspace is represented on the agenda by the smallest number of volumes. The effect of small vs. large versus large visibility volumes on the effectiveness of A* search should be investigated further.

Simplicity of the search function is the second significant advantage of this approach to the problem. Almost all of the work necessary to do a search is done before the search starts. No complex (and time consuming) pruning functions are implemented during the search. The only work not done before the search is the actual calculation of the cost and evaluation functions.

LIST OF REFERENCES

1. Deo, Narsingh and Pang, Chi-yen, "Shortest-Path Algorithms: Taxonomy and Annotation," Networks, Volume 14 (1984), John Wiley and Sons.

2. Rowe, Neil C., Artificial Intelligence Through PROLOG, Prentice Hall Inc., 1988.

3. Naval Postgraduate School Report Number NPS52-86-021, Solving Global Two Dimensional Routing Problems Using Snell's Law and A* Search by Neil C. Rowe, Micheal J. Zyda, and Robert B. McGhee, October 1986.

4. Richbourg, Robert F., Solving a Class of Spatial Reasoning Proglems: Minimal-cost Path Planning in the Cartesian Plane, Ph. D. Dissertation, Naval Postgraduate School, Monterey, California, June 1987.

5. Naval Postgraduate School Report Number NPS52-87-027, Roads, Rivers and Rocks: Optimal Path Planning Around Linear Features for a Mobile Agent by Neil C. Rowe, June 1987.

6. Sharir, Micha and Schorr, Amir, "On Shortest Paths in Polyhedral Spaces," Siam Journal of Computing, Volume 15, Number 1, February, 1986.

7. Alexander, Robert S., "Path Planning by Optimal-Path-Map Construction," Ph. D. Dissertation, Naval Postgraduate School, Monterey, California, to be published September 1989.

8. Hecht, Eugene and Zajac, Alfred, "Optics," Addison-Wesley Publishing Company, 1979.

9. Yee, Seung Hee, Three Algorithms for Planar-path Terrain Modeling, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.

10. Crow, Frankin C., "Shadow Algorithms for Computer Graphics," Computer Graphics, Volume 11, Number 2, Summer 1977.

11. Berkey, Dennis D., Calculus, CBS College Publishing, 1984.

12. Poulos, Dennis Duane, Range Image Processing for Local Navigation of an Autonomous Land Vehicle, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1986.

13. Preperata, Franco P., and Shamos, Michael Ian, Computational Geometry, an Introduction, Springer-Verlag, 1985.

14. Ross, Ron S., "Planning Minimum-Energy Paths in an Off-Road Environment with Anisotropic Traversal Costs and Motion Constraints," Ph. D. Dissertation, to be published June 1989.

15. Nicodemi, Olympia, Discrete Mathematics, West Publishing Company, 1987.

16. Qwak, Sehung, unpublished LISP graphics display program, May 1988.

17. Charniak, Eugene and McDermott, Drew, Introduction to Artificial Intelligence, Addison-Wesley Publishing Company, 1987.

18. Ralston, Anthony, A First Course in Numerical Analysis, McGraw-Hill Inc., 1965.

BIBLIOGRAPHY

Bromley, Hand and Lamson, Richard, LISP Lore: A Guide to Programming the LISP Machine, 2d ed., Kluwer Academic Publishers, 1987.

Clocksin, W. F., and Mellish, C. S., Programming in PROLOG, 3rd ed., Springer-Verlag, 1987.

Grossman, Stanley I., Multivariable Calculus, Linear Algebra, and Differential Equations, 2d ed., Academic Press Inc., 1986.

Naval Postgraduate School Report Number NPS52-87-003, A New Method for Optimal Path Planning Through Nonhomogeneous Free Space by Neil C. Rowe and R. F. Richbourg, February 1987.

Wilensky, Robert, Common LISPcraft, W. W. Norton and Company, 1986.

APPENDIX A. SAMPLES

This appendix contains a sample program run for a single ridge terrain model (see Chapter V, Section A) with one observer, and the contents of two terrain data files. The program run shows all user inputs and program responses from initial terrain selection through path planning (with a single optimization cycle).

Sample run:

```
> (set-up 1 "t20-ridge-x.lisp")

**********************************************************************
* Volume Creation and Display                          V1.1 *
**********************************************************************

>Constants Initialized
>Camera built
>Universe created; reading data file

>>>> Volume created: |volume0002|  Composition: GROUND
>>>> Volume created: |volume0003|  Composition: AIR

>Creation complete.



Find all ridges in ground terrain:

Ridge check, line: |line0022|
Ridge check, line: |line0021|
Ridge check, line: |line0020|
Ridge check, line: |line0019|
Ridge check, line: |line0018|
Ridge check, line: |line0017|
Ridge check, line: |line0016|
Ridge check, line: |line0015|
Ridge check, line: |line0014|
Ridge check, line: |line0013|
Ridge check, line: |line0012|
Ridge check, line: |line0011|
Ridge check, line: |line0010|
Ridge check, line: |line0009|
Ridge check, line: |line0008| -- Ridge
Ridge check, line: |line0007|
Ridge check, line: |line0006|
Ridge check, line: |line0005|
Ridge check, line: |line0004|
Ridge check, line: |line0003|
Ridge check, line: |line0002|



Making air volumes convex:

Air volumes: ((|volume0003|))
```

Ridge planes: (|plane0013|)

intersecting: (|volume0003| (1/500 0 0 1)) --- Result: (|facet0017|)

Re-doing error intercepts: NIL

Enter observer data now:

NIL
> (init-observer '(0 500 300) '0.75)
|observer0002|
> (set-up-2)

----------PRE-PROCESS VISIBILITY INFORMATION----------

making visibility regions for: |observer0002|

Air volumes: ((|volume0005|) (|volume0004|))
Limiting planes of visibility: (|plane0018|)

intersecting: (|volume0005| (-1/750 0 1/300 1)) --- Result: (|facet0023|)
intersecting: (|volume0004| (-1/750 0 1/300 1)) --- Result: (|facet0028|)

Re-doing error intercepts: NIL

Numeric errors: NIL

*******SPEED-DEMON-INVOKED*******

Visibility determination for: |observer0002|

|volume0006| visible
|volume0007| visible
|volume0009| not visible
|volume0008| visible

Determine Probability of Detection for visibility volumes

|volume0006| has P.D.: 0.75

```
|volume0007| has P.D.: 0.75
|volume0008| has P.D.: 0.75
|volume0009| has P.D.: 0.0
|volume0002| has P.D.: 0.0


*******SPEED-DEMON-INVOKED*******


Connecting volumes:

|volume0006| Connected to: (|volume0007| |volume0008|)
|volume0007| Connected to: (|volume0006|)
|volume0008| Connected to: (|volume0009| |volume0006|)
|volume0009| Connected to: (|volume0008|)
|volume0002| Connected to: NIL


NIL
> (a-star-search (init-point '(1000 1000 300)) (init-point '(0 0 300)) 'nil)

>>>>Begin A-star Search

    Start Volume: |volume0009|
    Goal Volume: |volume0006|


Search.Completed

>>>>Path Statistics:

  Maximum detection probability: 0.75
  Average detection probability: 0.4802796
  Total length of path: 1764.1292
  Total number of maneuvers: 2

>>>>Path: |path0002|

|path0002|
> (optimize-path '|path0002|)

Optimizing path |path0002|:

Optimizing at facet number 1 : |facet0028|
Optimizing at facet number 2 : |facet0017|


Optimization completed
>>>>Path Statistics:
```

Maximum detection probability: 0.75
Average detection probability: 0.49770224
Total length of path: 1533.2666
Total number of maneuvers: 2

>>>>Path: |path0003|

NIL

Sample of terrain using the first input method:

```
;
;*****************************************************************************
;
;
; ridge-x.lisp: a ridge oriented parallel to the y-axis (along a plane of the x-axis)
;
; USES FIRST INPUT METHOD
;
; D.H.Lewis/Thesis
;*****************************************************************************
;

((ground nil)                    ; terrain
 ((0 0 300)(0 1000 300))
 ((0 0 300)(300 0 300))
 ((300 0 300)(300 1000 300))
 ((300 1000 300)(0 1000 300))
 ((300 1000 300)(500 1000 500))
 ((500 1000 500)(700 1000 300))
 ((500 1000 500)(500 0 500))
 ((500 0 500)(300 0 300))
 ((500 0 500)(700 0 300))
 ((700 0 300)(700 1000 300))
 ((700 0 300)(1000 0 300))
 ((1000 0 300)(1000 1000 300))
 ((1000 1000 300)(700 1000 300))

 ((0 0 0)(0 1000 0))             ; structural lines for ground volume
 ((0 0 0)(0 0 300))
 ((0 0 0)(1000 0 0))
 ((0 1000 0)(0 1000 300))
 ((1000 0 0)(1000 0 300))
 ((0 1000 0)(1000 1000 0))
 ((1000 1000 0)(1000 1000 300))
 ((1000 1000 0)(1000 0 0)))

((air 0.01)                      ; terrain
 ((0 0 300)(0 1000 300))
 ((0 0 300)(300 0 300))
 ((300 0 300)(300 1000 300))
 ((300 1000 300)(0 1000 300))
 ((300 1000 300)(500 1000 500))
 ((500 1000 500)(700 1000 300))
 ((500 1000 500)(500 0 500))
 ((500 0 500)(300 0 300))
 ((500 0 500)(700 0 300))
 ((700 0 300)(700 1000 300))
 ((700 0 300)(1000 0 300))
 ((1000 0 300)(1000 1000 300))
 ((1000 1000 300)(700 1000 300))
```

```
((0 0 1000)(0 1000 1000))          ; structural line for air volume
((0 0 1000)(0 0 300))
((0 0 1000)(1000 0 1000))
((0 1000 1000)(0 1000 300))
((1000 0 1000)(1000 0 300))
((0 1000 1000)(1000 1000 1000))
((1000 1000 1000)(1000 1000 300))
((1000 1000 1000)(1000 0 1000)))
```

Sample of terrain using the second input method:

```
;
;*****************************************************************
;
; Double peak.lisp: Two adjacent ideal peaks
;
; USES SECOND INPUT METHOD
;
; D.H. Lewis        Thesis              11/12/88
;*****************************************************************
;


((((0 0 200)
   (500 250 500)
   (1000 0 200))
  ((0 0 200)
   (500 250 500)
   (0 500 200))
  ((0 500 200)
   (500 250 500)
   (1000 500 200))
  ((1000 500 200)
   (500 250 500)
   (1000 0 200)))
 (((0 500 200)
   (500 750 700)
   (1000 500 200))
  ((0 500 200)
   (500 750 700)
   (0 1000 200))
  ((0 1000 200)
   (500 750 700)
   (1000 1000 200))
  ((1000 1000 200)
   (500 750 700)
   (1000 500 200))))
```

FILE NAME: Setup.lisp

```
;*****************************************************************************
;                    MAIN  CONTROL  FUNCTIONS
; This file contains several LISP functions perform overall control of the
; static construction phase of the code.  They include the initial input control loops (for
; both input methods, and the control loop for the visibility region
; construction phase of the static construction.  The initial set-up
; functions are first, followed by the middle phase set-up functions, large
; scale control functions, and finally, the actual input methods themselves.
;
; THESIS                                D. H. LEWIS  20 OCT 88
;*****************************************************************************
;
;
;     ROUTINE TO INPUT A DATA STREAM AND CONSTRUCT THE VOLUME(S)
;
;  THESIS/CS4452                        D.H. LEWIS   15 MAY 88
;     ------------------------------------------------------
;
;  Builds the standard static flavors (Universe, above, below, and cameras),
; opens and reads input file, and carries the static phase through making
; air volumes convex.
;
; MAIN FUNCTIONS:  SET-UP (METHOD FILE)
;             SET-UP-2
;
; OTHER FUNCTIONS:  INITIALIZE-VOLUME
;             MAKE-VOLUME-WITH-FACET-DATA
;             DECREASSING-SORT-ON-X-P
;             DECREASING-SORT-ON-Y-P
;             PRINT-HEADER-1
;
;*****************************************************************************


(defvar *Universe*)          ; location of names for all flavors used in static
                             ; construction
(defvar *above*)             ; standard volumes used by intercept functions
(defvar *below*)             ;
(defvar *input-stream*)       ; system name for non-standard input file
(defvar *output-stream*)      ; system name for non-standard output file
(defvar *max-altitude* '1000)  ; maximum altitude in Input Method 2
(defvar *min-altitude* '0)     ; minimum altitude for Input Method 2
(defvar *not-convex-volumes*)  ; flag variable for Input Method 2 which tells
                             ; which facet building function(s) to use

(defvar *original-input-volumes* 'nil)   ; save various "states"
(defvar *convex-volumes* 'nil)
(defvar *final-visibility-regions* 'nil)
```

APPENDIX B. LISP CODE LISTING

This appendix contains the listings for all LISP code in the files shown in Table 2. File names are noted at the beginning of each file listing.

```
;----------INITIAL SETUP----------

(defun set-up (Method File)
 (print-header-1)
 (make-origin)                          ; make favorite constants
 (make-null-vector)
 (setf *above* (make-instance 'volume))
 (setf *below* (make-instance 'volume))
 (setf *not-convex-volumes* 't)
 (setf *done-making-new-visibility-volumes-flag* 'nil)
 (setf *precision* '0.0025)
 (setf *large-integer* '1e4)
 (setf *list-of-error-planes* 'nil)
 (princ ">Constants Initialized") (terpri)
 (make-cameras)                         ; create camera
 (princ ">Camera built") (terpri)
 (setf *Universe* (make-instance 'Universe       ; create universe for volumes

                                  :Volumes '()
                                  :observers '()))
 (princ ">Universe created; reading data file") (terpri) (terpri)
 (setq *input-stream* (open File :direction :input))

                    ; select and use input method

 (cond ((equal '1 Method)       ; volume oriented input method
        (do ((data (read *input-stream* nil 'done) ; read all volumes into universe
                   (setf data (read *input-stream* nil 'done))))
            ((atom data))
          (push (init-volume data) (universe-volumes *Universe*))
          (princ ">>>> Volume created: ")
          (prin1 (car *list-of-volumes*))
          (princ "   Composition: ")
          (prin1 (volume-composition (eval (car *list-of-volumes*)))) (terpri)
          (make-all-facets (car *list-of-volumes*))) ; make all facets for new volume
        (loop for V in (universe-volumes *universe*)
          do (setf (volume-visibility (eval V)) 'nil)))  ; remove ambiant visibility

       ((equal '2 Method)       ; facet oriented input method
         (do ((data (read *input-stream* nil 'done) ; read all volumes into universe
                    (setf data (read *input-stream* nil 'done))))
             ((atom data))
           (loop for terrain-segment in data      ; go through each volume segment
             do (setf (universe-volumes *universe*)
                      (append (make-volume-with-facet-data terrain-segment)
                              (universe-volumes *universe*))))))
       (t (terpri) (princ "Error: Method not implemented")
          (return-from set-up 'Done)))

 (close *input-stream*) (terpri)
```

```
 (princ ">Creation complete.") (terpri) (terpri)
 (setf *original-input-volumes* (universe-volumes *universe*))

                    ; complete initial setup functions

 (find-all-ridges)
 (terpri)
 (make-convex-volumes)
 (setf *not-convex-volumes* 'nil)
 (setf *convex-volumes* (universe-volumes *universe*))
 (terpri) (terpri) (princ "Enter observer data now: ") (terpri) (terpri))


(defun print-header-1 ()
 (terpri)
 (terpri)
 (princ "**************************************************************************")
 (terpri)
 (princ "* Volume Creation and Display                    V1.1  *")
 (terpri)
 (princ "**************************************************************************")
 (terpri)
 (terpri))


;---------------------------INPUT METHOD ONE-----------------------------------


(defun Initialize-volume (Volume Data)      ; expects data as:
 (cond                                      ; (line line line ....) where lines are
   ((null Data) Volume)                     ; (point point)  where points are; (x y z).
                                            ; (((x y z) (x y z)) ((x y z) (x y z)))
   (t (create-new-line Volume (init-point (caar Data)) (init-point (cadar Data)))
     (Initialize-volume Volume (cdr Data)))))



(defun create-new-line (Volume pt1 pt2)     ; put points and lines into volume instance
 (pushnew pt1 (volume-points (eval Volume)))
 (pushnew pt2 (volume-points (eval Volume)))
 (pushnew (init-Line (init-vector '*origin* pt1) (init-vector pt1 pt2))
                                    (volume-Edges (eval Volume))))



;---------------------------INPUT METHOD TWO-----------------------------------


(defun make-volume-with-facet-data (data)  ; construct a volume from a formatted
                        ; list of data where format is:
                        ; (facet facet facet...)
```

```
(let ((terrain-facets (build-terrain data))
      (terrain-box (make-instance 'bounding-box))  ; used to find limits of data
                                                   ; points
      (ground-volume (init-volume (list (list 'ground 'nil))))
      (air-volume (init-volume (list (list 'air 'nil))))
      (points-and-lines 'nil))

            ; find all lines and points in terrain facets

 (setf points-and-lines (info-on-facets terrain-facets))

            ; assign values to air and ground volumes

 (setf (volume-composition (eval ground-volume)) 'ground)   ; set composition
 (setf (volume-composition (eval air-volume)) 'air)
 (setf (volume-facets (eval ground-volume)) terrain-facets)  ; put terrain facets
 (setf (volume-facets (eval air-volume)) terrain-facets)

            ; construct top/bottom and sides of ground and air volumes

 (send terrain-box :construct-bounding-box (first points-and-lines))
 (let ((point-1 (first (find-point (bounding-box-max-x terrain-box);corners of terrain
                                   (bounding-box-min-y terrain-box)
                                   'nil
                                   (first points-and-lines))))
       (point-2 (first (find-point (bounding-box-max-x terrain-box)
                                   (bounding-box-max-y terrain-box)
                                   'nil
                                   (first points-and-lines))))
       (point-3 (first (find-point (bounding-box-min-x terrain-box)
                                   (bounding-box-max-y terrain-box)
                                   'nil
                                   (first points-and-lines))))
       (point-4 (first (find-point (bounding-box-min-x terrain-box)
                                   (bounding-box-min-y terrain-box)
                                   'nil
                                   (first points-and-lines))))
       (points-41 (stable-sort (find-point 'nil          ; edges of terrain
                                           (bounding-box-min-y terrain-box)
                                           'nil
                                               (first points-and-lines))
                               #'decreasing-sort-x-p))
       (points-12 (stable-sort (find-point (bounding-box-max-x terrain-box)
                                           'nil
                                           'nil
                                               (first points-and-lines))
                               #'decreasing-sort-y-p))
       (points-23 (stable-sort (find-point 'nil
                                           (bounding-box-max-y terrain-box)
                                           'nil
                                               (first points-and-lines))
```

```
                                #'decreasing-sort-x-p))
        (points-34 (stable-sort (find-point (bounding-box-min-x terrain-box)
                                            'nil
                                            'nil
                                                (first points-and-lines))
                                #'decreasing-sort-y-p))
        (top-points 'nil)                       ; top and bottom
        (bottom-points 'nil))                     ; points

(loop for P in (list point-1 point-2 point-3 point-4)      ; find top points
      do (setf top-points (adjoin (init-point (list
                                                  (first
                                                    (send (eval P) :list-format))
                                                (second
                                                    (send (eval P) :list-format))
                                                *max-altitude*))
                                  top-points)))
(setf top-points (reverse top-points))
(setf (volume-facets (eval air-volume))                 ; make top facet
        (adjoin (make-a-facet top-points)
                (volume-facets (eval air-volume))))

(setf (volume-facets (eval air-volume))                 ; make top sides
        (adjoin (build-side-facet (fourth top-points)       ; and put in volume
                                  (first top-points)
                                  points-41)
                (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
        (adjoin (build-side-facet (first top-points)
                                  (second top-points)
                                  points-12)
                (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
        (adjoin (build-side-facet (third top-points)
                                  (second top-points)
                                  points-23)
                (volume-facets (eval air-volume))))
(setf (volume-facets (eval air-volume))
        (adjoin (build-side-facet (fourth top-points)
                                  (third top-points)
                                  points-34)
                (volume-facets (eval air-volume))))

(loop for P in (list point-1 point-2 point-3 point-4)      ; find bottom points
      do (setf bottom-points (adjoin (init-point (list
                                                   (first (send (eval P) :list-format))
                                                 (second (send (eval P) :list-format))
                                                 *min-altitude*))
                                   bottom-points)))
(setf bottom-points (reverse bottom-points))
(setf (volume-facets (eval ground-volume))              ; make bottom facet
```

```
            (adjoin (make-a-facet bottom-points)
                    (volume-facets (eval ground-volume))))

    (setf (volume-facets (eval ground-volume))              ; make bottom sides
            (adjoin (build-side-facet (fourth bottom-points)    ; and put in volume
                                      (first bottom-points)
                                      points-41)
                    (volume-facets (eval ground-volume))))
    (setf (volume-facets (eval ground-volume))
            (adjoin (build-side-facet (first bottom-points)
                                      (second bottom-points)
                                      points-12)
                    (volume-facets (eval ground-volume))))
    (setf (volume-facets (eval ground-volume))
            (adjoin (build-side-facet (third bottom-points)
                                      (second bottom-points)
                                      points-23)
                    (volume-facets (eval ground-volume))))
    (setf (volume-facets (eval ground-volume))
            (adjoin (build-side-facet (fourth bottom-points)
                                      (third bottom-points)
                                      points-34)
                    (volume-facets (eval ground-volume)))))

                     ; complete information on volumes

   (setf points-and-lines (info-on-facets (volume-facets (eval air-volume))))
   (setf (volume-points (eval air-volume)) (first points-and-lines))
   (setf (volume-edges (eval air-volume)) (second points-and-lines))
   (setf points-and-lines (info-on-facets (volume-facets (eval ground-volume))))
   (setf (volume-points (eval ground-volume)) (first points-and-lines))
   (setf (volume-edges (eval ground-volume)) (second points-and-lines))
   (loop for V in (list air-volume ground-volume)
    do (let ()
            (terpri) (princ ">>>> Volume Created: ")
            (prin1 V) (princ " Composition: ") (prin1 (volume-composition (eval V)))))
   (list air-volume ground-volume)))

(defun decreasing-sort-x-p (A B)
 (cond ((> (first (send (eval A) :list-format))
           (first (send (eval B) :list-format))))))

(defun decreasing-sort-y-p (A B)
 (cond ((> (second (send (eval A) :list-format))
           (second (send (eval B) :list-format))))))



;*****************************************************************************
;
;
;       COMPLETE  STATIC  PHASE
```

```
;      ------------------------------------------------------------
;
;  Functions here complete the static phase of construction of the visibility
; regions.
;
; MAIN FUNCTIONS: SET-UP-2
;
;*****************************************************************************
;

(defun set-up-2 ()    ; finish initial setup (after observers created)
  (let ((observers (universe-observers *universe*)))
    (terpri) (terpri)
    (princ "------------PRE-PROCESS VISIBILITY INFORMATION------------")
    (terpri) (terpri)
    (loop for Obs in observers                    ;divide up universe by visibilities
      do (make-visibility-regions Obs))
    (terpri) (terpri)
    (princ "Numeric errors: ") (prin1 *list-of-error-planes*)
    (terpri) (terpri)
    (send *universe* :save-static-items)
    (setf *final-visibility-regions* (universe-volumes *universe*))
    (setf *done-making-new-visibility-volumes-flag* 't)  ; speed things up
    (loop for Obs in observers                    ; find who can see what
      do (speed-demon)
      do (determine-visibility Obs))
    (terpri)(terpri)
    (princ "Determine Probability of Detection for visibility volumes")
    (terpri)
    (loop for V in (universe-volumes *universe*)     ; calc prob of detection for
      do (probabilities-assuming-independence-or V)) ; each volume
    (terpri) (terpri)
    (speed-demon)
    (connect-volumes)                             ; build visibility graph
    (terpri)))
```

FILE NAME: Volume-functions.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;;;*****************************************************************
;;;
;;; D.H. Lewis          CS4452/THESIS          5May88
;;;
;;;*****************************************************************
;
;                  FLAVORS AND METHODS
;
;          ---------------------------------------
;
; FLAVOR:  .....Point
;
; METHODS:  :LIST-FORMAT              ; give the x,y and z values as a three-tuple
;           :LIST-FORMAT-REAL         ; same, in real number format
;           :LIST-FORMAT-4            ; give agraphics 4-tuple "(x y z 1)"
;           :PRINT                    ; print info on point
;
; FLAVOR:  .....Vector
;
; METHODS:  :LENGTH                   ; calculate length of vector
;           :UNIT-VECTOR              ; return the coeff of the unit vector
;           :ENDPOINTS                ; give the names of the endpoints of the vector
;           :LIST-FORMAT              ; give the coeffs of the vector as a 3-tuple
;           :LIST-FORMAT-REAL         ; same, execept with real numbers
;           :PRINT                    ; print coeff values to output file
;
; FLAVOR:  .....Line-segment
;
; METHODS:  :ENDPOINTS                ; Return the endpoints of the line-segment
;           :ENDPOINT-LIST            ; Return endpoints as explicit 4-tuples
;           :OTHER-END (ENDPOINT)     ; Given one endpoint, return the other
;           :START-POINT              ; Return the start point of the line-segment
;           :END-POINT                ;  "   " end point  " "   "
;           :LENGTH                   ; Find and return the length of the
;                                       line-segment
;           :BACKSUBS (T-LIST)        ; Subsitute the (Tx Ty Tz) list into
;                                       the line equation
;           :MID-POINT                ; Find the mid point of the line-segment
;           :STRATTLE-PLANE-P (PLANE) ; do the endpoints of the line-segment
;                                       lie on opposite sides of the plane?
;           :PRINT
;
; FLAVOR:  .....Plane
;
; METHODS:  :TEST-EQUAL (PLANE)  ; Do two planes have the same coeffs?
;           :LIST-COEFF          ; List 4-tuple (X Y Z Ao) for plane
;           :LIST-COEFF-3        ; List 3-tuple (X Y Z) for plane
;           :SUBS-POINT-INTO-PLANE (POINT)     ; Get Ao coeff from X,Y,Z values of point
```

```
;       :FIND-Z-GIVEN-XY (X Y)          ; Return Z value of point in plane
;       :FIND-Y-GIVEN-XZ (X Z)          ;  "   X  "  "  "  "  "
;       :FIND-X-GIVEN-YZ (Y Z)          ;  "   Y  "  "  "  "  "
;       :PRINT
;
; FLAVOR:  .....Bounding-box
;
; METHODS:  :CONSTRUCT-BOUNDING-BOX (POINTS)     ; Build a 3-D limits for list of
points
;       :INSIDE-BOUNDING-BOX (POINT)      ; Is the point inside the limits?
;
; FLAVOR:  .....Facet
;
; METHODS: :POINTS
;       :PRINT
;
; FLAVOR:  .....Volume
;
; METHODS: :MAKE-EQUAL
;       :CLEAR
;       :FIND-ARITHMETIC-CENTER
;       :MAKE-NODE-LIST
;       :MAKE-POLYGON-LIST
;       :PRINT
;
; FLAVOR:  .....Universe
;
; METHODS: :SAVE-STATIC-ITEMS
;       :REVERT-STATIC-ITEMS
;*****************************************************************

(defvar *origin*)
(defvar *null-vector*)
(defvar *one-vector* '(1.0 1.0 1.0 1.0))
(defvar *one-vector-3* '(1.0 1.0 1.0))
(defvar *zero-vector* '(0.0 0.0 0.0 0.0))
(defvar *zero-vector-3* '(0.0 0.0 0.0))
(defvar *max-counter-value* '9999)
(defvar *done-making-new-visibility-volumes-flag* 'nil)

(defvar *list-of-points* 'nil)
(defvar *points-counter* '0)
(defvar *minimum-points-counter* '0)

(defvar *list-of-vectors* 'nil)
(defvar *vectors-counter* '0)
(defvar *minimum-vectors-counter* '0)

(defvar *list-of-lines* 'nil)
(defvar *lines-counter* '0)
(defvar *minimum-lines-counter* '0)
```

```
(defvar *list-of-planes* 'nil)
(defvar *planes-counter* '0)
(defvar *minimum-planes-counter* '0)

(defvar *list-of-facets* '())
(defvar *facets-counter* '0)
(defvar *minimum-facets-counter* '0)

(defvar *list-of-volumes* '())
(defvar *volumes-counter* '0)
(defvar *minimum-volumes-counter* '0)


;------POINT------

(defflavor point
        (x-coord
         y-coord
         z-coord)
        ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables)

(defmethod (point :List-format) ()  ; return a 3-tuple "(X Y Z)"
 (list x-coord y-coord z-coord))

(defmethod (point :List-format-real) () ; return a real valued 3-tuple
 (map 'list '* (list x-coord y-coord z-coord) (make-list 3 :initial-element '1.0)))

(defmethod (point :List-format-4) ()  ; return list in graphics format
 (list x-coord y-coord z-coord '1))

(defmethod (point :print) ()
 (pprint (list x-coord y-coord z-coord) *output-stream*))



;-------VECTOR------

(defflavor vector
        (i
         j
         k
         Start-point
         End-point)
        ()
  :gettable-instance-variables
  :settable-instance-variables
```

```
   :inittable-instance-variables
   :outside-accessible-instance-variables)

(defmethod (vector :length) () ; Calculate the length of a vector
  (sqrt (abs (+ (* i i) (* j j) (* k k)))))

(defmethod (vector :unit-vector) () ; make a unit vector from a vector
  (let ((vector-length (send self :length)))
    (cond ((equal-zero-p vector-length) '(0.0 0.0 0.0))
          (t (map 'list '/ (send self :list-format)
                           (make-list 3 :initial-element vector-length))))))

(defmethod (vector :endpoints) () ; find the endpoints of the vector
  (list Start-point End-point))

(defmethod (vector :list-format) () ; return the values of the vector as a 3-tuple
  (list i j k))

(defmethod (vector :list-format-real) () ; return a real valued 3-tuple
  (map 'list '* (list i j k) (make-list 3 :initial-element '1.0)))

(defmethod (vector :print) ()
  (pprint (list i j k Start-point End-point) *output-stream*))



;--------LINE SEGMENT-------

(defflavor line-segment          ; position vector can point to either end of
          (t-max                 ; direction vector. direction vector can point
           position-vector       ; in either direction between endpoints
           direction-vector
           characteristics)      ; ridge, valley, etc
          ()
   :gettable-instance-variables
   :settable-instance-variables
   :inittable-instance-variables
   :outside-accessible-instance-variables)

(defmethod (line-segment :endpoints) ()  ;get endpoints of the line segment
   (send (eval direction-vector) :endpoints))

(defmethod (line-segment :endpoint-list) () ; get endpoints in graphics format
   (list (send (eval (car (send self :endpoints))) :list-format-4)
         (send (eval (cadr (send self :endpoints))) :list-format-4)))

(defmethod (line-segment :other-end) (endpoint) ; find the endpoint of the line-segment
  (let ((line-endpoints (send self :endpoints))) ; opposite of the given endpoint
    (cond ((equal endpoint (first line-endpoints))
           (second line-endpoints))
          (t (first line-endpoints)))))
```

```
(defmethod (line-segment :start-point) () ; what is the start point of the line-segment?
 (vector-start-point (eval direction-vector)))

(defmethod (line-segment :end-point) () ; what is the end point of the line segment?
 (vector-end-point (eval direction-vector)))

(defmethod (line-segment :length) ()  ; how long is the line-segment?
 (send (eval direction-vector) :length))

(defmethod (line-segment :backsubs) (t-list)   ; subs a list of t-parameters
                                ; back into the line equation to get
                                ; the (x y z) coord of the point
 (mapcar '+ (send (eval position-vector) :list-format-real)
            (mapcar '* t-list
                    (send (eval direction-vector) :list-format-real))))

(defmethod (line-segment :midpoint) ()
 (let ((t-half (/ t-max '2.0)))
  (send self :backsubs (list t-half t-half t-half))))

(defmethod (line-segment :strattle-plane-p) (plane)
                        ; return T iff the endpoints of self
                        ; are on opposite sides of the given plane
 (let ((Ao-1 (send (eval plane) :point-into-equation
                                    (first (send self :endpoints))))
       (Ao-2 (send (eval plane) :point-into-equation
                                    (second (send self :endpoints))))
       (Ao (fourth (send (eval plane) :list-coeff))))
  (cond ((or (equal-error Ao Ao-1)
             (equal-error Ao Ao-2))
         'nil)
        ((or (and (GE Ao-1 Ao)
                  (LE Ao-2 Ao))
             (and (LE Ao-1 Ao)
                  (GE Ao-2 Ao)))
         't))))

(defmethod (line-segment :print) ()
 (pprint t-max *output-stream*)
 (pprint (list position-vector (send (eval position-vector) :list-format)
                  (send (eval position-vector) :endpoints)) *output-stream*)
 (pprint (list direction-vector (send (eval direction-vector) :list-format)
                  (send (eval direction-vector) :endpoints)) *output-stream*)
 (pprint (send self :endpoints) *output-stream*)
 (pprint characteristics *output-stream*))


;-------PLANE-------
```

```
(defflavor plane                ; uses equation of plane:
        (a-coef                 ;
         b-coef                 ; aX + bY + cZ = Ao
         c-coef                 ;
         Ao)                    ; for comparisions, equation is generally
        ()                      ; normalized, so Ao=-1,+1 or 0.
  :gettable-instance-variables      ; NOTE: first non-zero coeff will ALWAYS be a
  :settable-instance-variables      ; positive number. Avoids direction ambiguity
  :inittable-instance-variables
  :outside-accessible-instance-variables)

(defmethod (plane :test-equal) (F2)  ; test plane for equality by comparing
                                     ; coefficients, or comparing the coeffs
                                     ; of the unit normal vectors
 (let ((V1 (init-vector '*origin* (init-point (send self :list-coeff-3))))
       (V2 (init-vector '*origin* (init-point (send (eval F2) :list-coeff-3)))))
  (or (apply 'and
             (map 'list #'equal-error
                  (send self :list-coeff)
                  (send (eval F2) :list-coeff)))
      (apply 'and
             (map 'list #'equal-error
                  (send (eval V1) :unit-vector)
                      (send (eval V2) :unit-vector))))))

(defmethod (plane :list-coeff) ()  ; list plane coefficents as a 4-tuple
 (list a-coef b-coef c-coef Ao))  ; (includes the Ao constant term)

(defmethod (plane :list-coeff-3) ()  ; list only the x,y,z coefficents
 (list a-coef b-coef c-coef))

(defmethod (plane :subs-point-into-plane) (Pt)  ; subs a point into the planar
                                    ; equation, returns result.
 (apply '+ (map 'list '* (send self :list-coeff-3) (send (eval Pt) :list-format))))

(defmethod (plane :point-into-equation) (point)  ; subs point into plane equation
                                    ; same as above  *****REMOVE******
 (apply '+ (map 'list '* (send (eval point) :list-format)
                         (send self : list-coeff-3))))

(defmethod (plane :find-x-given-yz) (y z)  ; find the x value of a point given the
 (cond ((equal-zero-p a-coef) '0)  ; y and z coordinates of a point, for
                                   ; the plane under consideration
       (t (/ (- Ao (+ (* b-coef y) (*  c-coef z))) a-coef))))

(defmethod (plane :find-y-given-xz) (x z)  ; find the y value of a point given the
 (cond ((equal-zero-p b-coef) '0)  ; x and z coordinates of a point, for
                                   ; the plane under consideration
       (t (/ (- Ao (+ (* a-coef x) (*  c-coef z))) b-coef))))

(defmethod (plane :find-z-given-xy) (x y)  ; find the z value of a point given the
```

```
 (cond ((equal-zero-p c-coef) '0)    ; x and y coordinates of a point, for
                                     ; the plane under consideration
       (t (/ (- Ao (+ (* a-coef x) (*  b-coef y))) c-coef))))

(defmethod (plane :print) ()
 (pprint (send self :list-coeff) *output-stream*))



;---------BOUNDING BOX---------

(defflavor Bounding-box          ; generalized bounding box flavor
          (max-x
           min-x
           max-y
           min-y
           max-z
           min-z)
          ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables
  :required-methods)

(defmethod (bounding-box :construct-bounding-box) (points)
                                  ; build bounding box for
                                  ; a list of points
 (let* ((first-point (send (eval (first points)) :list-format))
          (x (first first-point))
          (y (second first-point))
          (z (third first-point)))
  (setf max-x x)
  (setf min-x x)
  (setf max-y y)
  (setf min-y y)
  (setf max-z z)
  (setf min-z z)
  (loop for P in (rest points)
   do (let* ((next-point (send (eval P) :list-format))
             (new-x (first next-point))
             (new-y (second next-point))
             (new-z (third next-point)))
            (cond ((GT new-x max-x)
                     (setf max-x new-x))
                  ((LT new-x min-x)
                     (setf min-x new-x)))
            (cond ((GT new-y max-y)
                     (setf max-y new-y))
                  ((LT new-y min-y)
                     (setf min-y new-y)))
```

```
            (cond ((GT new-z max-z)
                   (setf max-z new-z))
                  ((LT new-z min-z)
                   (setf min-z new-z)))))

(defmethod (bounding-box :inside-bounding-box-p) (point)
                         ; return T if point is inside
                         ; bounding box, NIL otherwise
 (let ((p (map 'list '* (send (eval point) :list-format) '(1.0 1.0 1.0))))
  (cond ((and (and (GE max-x (first p))
                   (LE min-x (first p)))
              (and (GE max-y (second p))
                   (LE min-y (second p)))
              (and (GE max-z (third p))
                   (LE min-z (third p))))
         't)
        (t 'nil))))



;-------FACET-------

(defflavor facet
        (edges                ;list of all edges bounding facet
         center               ; location of center of facet
         connects)              ; volumes which facet connects "((V1..Vn) (V2..Vm))"
        (plane                ;mixin flavors
         bounding-box)
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables
  :required-methods)

(defmethod (facet :points) () ; return all verticies of facet
 (let ((temp 'nil))
  (loop for E in Edges
    do (setf temp (append temp (send (eval E) :endpoints))))
  (delete-duplicates temp)))

(defmethod (facet :find-facet-center) ()  ; find the average of all the vertices
                          ; of the facet.
 (let* ((points (send self :points))
        (temp-sum (send (eval (first points)) :list-format))
        (nr-points (length points)))
  (loop for P in (rest points)
   do (setf temp-sum (map 'list '+ temp-sum
                                  (send (eval P) :list-format))))
  (setf (facet-center self)
        (init-point (map 'list '/ temp-sum (make-list 3 :initial-element nr-points))))
  (facet-center self)))
```

```
(defmethod (facet :add-volume-to-left-connects) (V) ; add a volume to the left list
                                    ; of the connects variable
 (cond ((null (facet-connects self))
          (setf (facet-connects self) (list (list V))))
        ((not (member-p V (first (facet-connects self))))
          (setf (first (facet-connects self)) (adjoin V (first (facet-connects self)))))))

(defmethod (facet :add-volume-to-right-connects) (V) ; add a volume to the right list
                                          ; of the connects variable
 (cond ((equal '1 (length (facet-connects self)))
          (setf (facet-connects self) (list (first (facet-connects self)) (list V))))
        ((not (member-p V (second (facet-connects self))))
          (setf (second (facet-connects self)) (adjoin V (second (facet-connects self)))))))


(defmethod (facet :print) ()
 (pprint (list edges center connects (send self :list-coeff)) *output-stream*))



;------VOLUME------

(defflavor volume
          (Visibility              ; visible observers
           Probability-of-detection    ; sum of PD for observers
           Composition              ; ground, air, etc
           Points                   ; all vertices of the volume
           Edges                    ; all line-segments of the volume
           Facets                   ; all surfaces of the volume
           Arithmetic-center        ; numeric average of the points
           connected-to)            ; adjacent volumes
          (Graphic)                 ; for 3-D projection
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables
  :required-methods)

(defmethod (volume :make-equal) (new-volume-name)
                                          ; make a new volume with same instances
 (let ((temp new-volume-name))            ; as self
  (set temp (make-instance 'volume
                    :Visibility Visibility
                    :Probability-of-detection Probability-of-detection
                    :Composition Composition
                    :Points Points
                    :Edges Edges
                    :Facets Facets
                    :arithmetic-center Arithmetic-center
                    :connected-to Connected-to))))
```

```
(defmethod (volume :clear) ()  ; clear out old values of an existing volumes
 (setf Visibility 'nil)
 (setf Probability-of-detection 'nil)
 (setf Composition 'nil)
 (setf Points 'nil)
 (setf Edges 'nil)
 (setf Facets 'nil)
 (setf Arithmetic-center 'nil)
 (setf Connected-to 'nil))

(defmethod (volume :find-arithmetic-center) ()  ; find the average of all the vertices
                                    ; of the volume. do not change values
                                    ; in the volume
 (let ((temp-sum (send (eval (first Points)) :list-format))
       (nr-points (length Points)))
  (loop for P in (rest Points)
   do (setf temp-sum (map 'list '+ temp-sum
                                    (send (eval P) :list-format))))
  (init-point (map 'list '/ temp-sum (make-list 3 :initial-element nr-points)))))

(defmethod (volume :make-node-list) () ; make a list of absolute point coords in graphic
 (loop for P in points              ; format (eg 4 element list)
                                  ; used in GRAPHICS.
   collect (reverse (append (list '1) (reverse (send (eval P) :list-format))))))

(defmethod (volume :make-polygon-list) ()  ;index point values to points in node list
  (loop for L in edges              ; used in GRAPHICS
    do (setf Pt1 (car (send (eval L) :endpoint-list)))
    do (setf Pt2 (cadr (send (eval L) :endpoint-list)))
     collect (list (position-if '(lambda (A) (equal A Pt1)) node-list)
                   (position-if '(lambda (A) (equal A Pt2)) node-list))))

(defmethod (volume :print) ()
 (pprint (list Visibility Probability-of-detection Composition Points Edges Facets
               arithmetic-center connected-to) *output-stream*))


;------UNIVERSE-------

(defflavor Universe              ; space of all volumes
        (Volumes
         Observers               ; observers located within the defined universe
         static-vectors          ; save the state of the lines, points and
         static-vector-counter       ; vectors used to build the static visibility
         static-lines            ; model
         static-lines-counter
         static-points
         static-points-counter)
        ()
  :gettable-instance-variables
```

```
   :settable-instance-variables
   :inittable-instance-variables
   :outside-accessible-instance-variables)

(defmethod (universe :save-static-items) ()      ; save state of static universe
  (setf static-vectors *list-of-vectors*)
  (setf *minimum-vectors-counter* *vectors-counter*)
  (setf static-lines *list-of-lines*)
  (setf *minimum-lines-counter* *lines-counter*)
  (setf static-points *list-of-points*)
  (setf *minimum-points-counter* *points-counter*)
  (setf *minimum-planes-counter* *planes-counter*)
  (setf *minimum-facets-counter* *facets-counter*)
  (setf *minimum-volumes-counter* *volumes-counter*))


;*****************************************************************************
;
;;;
;;; FUNCTIONS TO INITIALIZE; GET NAMES OF OBJECTS AND MAKE NAMES GLOBAL
;;;
;*****************************************************************************
;



(defun make-origin ()                        ; names of special points and
  (gensym (incf *points-counter*))               ; other unique flavors.
  (setf *origin* (make-instance 'point
                                :x-coord '0
                                :y-coord '0
                                :z-coord '0))
  (pushnew '*origin* *list-of-points*))



(defun make-null-vector ()
  (gensym (incf *vectors-counter*))
  (setf *null-vector* (make-instance 'vector
                                     :i '0
                                     :j '0
                                     :k '0
                                     :Start-point '*origin*
                                     :End-point '*origin*))
  (push '*null-vector* *list-of-vectors*))



(defun make-point-name ()              ;produce variable names "on the fly"
  (cond ((> *points-counter* (1- *max-counter-value*))
         (setf *points-counter* *minimum-points-counter*)))
  (gensym (incf *points-counter*))
  (intern (gensym "point")))
```

```
(defun make-line-name ()
 (cond ((> *lines-counter* (1- *max-counter-value*))
        (setf *lines-counter* *minimum-lines-counter*)))
 (gensym (incf *lines-counter*))
 (intern (gensym "line")))

(defun make-vector-name ()
 (cond ((> *vectors-counter* (1- *max-counter-value*))
        (setf *vectors-counter* *minimum-vectors-counter*)))
 (gensym (incf *vectors-counter*))
 (intern (gensym "vector")))

(defun make-facet-name ()
 (cond ((> *facets-counter* (1- *max-counter-value*))
        (setf *facets-counter* *minimum-facets-counter*)))
 (gensym (incf *facets-counter*))
 (intern (gensym "facet")))

(defun make-plane-name ()
 (cond ((> *planes-counter* (1- *max-counter-value*))
        (setf *planes-counter* *minimum-planes-counter*)))
 (gensym (incf *planes-counter*))
 (intern (gensym "plane")))

(defun make-volume-name ()
 (cond ((> *volumes-counter* (1- *max-counter-value*))
        (setf *volumes-counter* *minimum-volumes-counter*)))
 (gensym (incf *volumes-counter*))
 (intern (gensym "volume")))

;*****************************************************************************
;
;;;
;;; FLAVOR INSTANTIATION FUNCTIONS
;;;
;;; Note: all of therse functions will stop keeping lists of previously
;;;       created instantiations after flag
;;;       *done-making-new-visibility-volumes-flag* is set to T
;*****************************************************************************
;


;-------MAKE A POINT--------

(defun init-point (List-of-values)   ; see if point already exists (nonrecursive)
 (cond ((and (not (null *list-of-points*))
             (not *done-making-new-visibility-volumes-flag*))
        (loop for P in *list-of-points*
          do (cond ((apply 'and
                           (map 'list #'equal-error
                                      (map 'list 'rationalize list-of-values)
                                          (send (eval P) :list-format)))
```

```
                    (return-from init-point P))))))
   (init-new-point list-of-values))

(defun init-new-point (List-of-values)
 (let ((temp (make-point-name)))
  (set temp (make-instance 'point
                           :x-coord (rationalize (first List-of-values))
                           :y-coord (rationalize (second List-of-values))
                           :z-coord (rationalize (third List-of-values))))
  (push temp *list-of-points*)
  temp))

;--------MAKE A VECTOR-------

(defun init-vector (Start-point End-point)  ; check to see if vector already built
 (cond ((not *done-making-new-visibility-volumes-flag*)
        (loop for V in *list-of-vectors*
          do (cond ((equal (send (eval V) :endpoints)
                           (list Start-point End-point))
                    (return-from init-vector V))))))
 (init-new-vector Start-point End-point))

(defun init-new-vector (Sp Ep)
 (let ((temp (make-vector-name)))
  (set  temp (make-instance 'vector
                            :i (- (point-x-coord (eval Ep)) (point-x-coord (eval Sp)))
                            :j (- (point-y-coord (eval Ep)) (point-y-coord (eval Sp)))
                            :k (- (point-z-coord (eval Ep)) (point-z-coord (eval Sp)))
                            :Start-point Sp
                            :End-point Ep))
  (push temp *list-of-vectors*)
  temp))



;--------MAKE A LINE SEGMENT-------

(defun init-line (Position-vector Direction-vector)  ; valid construction for a line???
        (cond ((and (equal (vector-Start-point (eval Position-vector)) '*origin*)
                    (member-p (vector-end-point (eval Position-vector))
                              (send (eval Direction-vector) :endpoints)))
               (Find-or-make-line  Position-vector Direction-vector))
              (t (terpri)
                 (princ "Error invalid vectors: ")
                 (prin1 (list position-vector direction-vector)) (terpri))))

(defun Find-or-make-line (Pv Dv)          ; check to see if line already built
 (cond ((not *done-making-new-visibility-volumes-flag*)
        (loop for L in *list-of-lines*
          do (cond ((and (member-p (vector-end-point (eval Pv))
                                   (send (eval (old-line-Dv L)) :endpoints))
```

```
                    (or (equal (send (eval Dv) :endpoints)
                               (send (eval (old-line-Dv L)) :endpoints))
                        (equal (send (eval Dv) :endpoints)
                            (nreverse (send (eval (old-line-Dv L)) :endpoints)))))
                (return-from find-or-make-line L))))))
 (init-new-line Pv Dv))

(defun init-new-line (Pv Dv)
 (let ((temp (make-line-name)))
  (set temp (make-instance 'line-segment
                           :t-max '1
                           :Position-vector Pv
                           :Direction-vector Dv
                           :characteristics 'nil))
  (push temp *list-of-lines*)
  temp))

(defun old-line-Dv (Line)
 (line-segment-Direction-vector (eval Line)))


;----------MAKE A PLANE---------


(defun init-plane (List-of-values)   ; see if plane already exists (nonrecursive)
 (cond ((and (not (null *list-of-planes*))
             (not *done-making-new-visibility-volumes-flag*))
        (loop for P in *list-of-planes*
          do (cond ((or (equal (send (eval P) :list-coeff)
                               list-of-values)
                        (apply 'and (map 'list #'equal-error
                                         (send (eval P) :list-coeff)
                                         list-of-values)))
                   (return-from init-plane P))))))
  (init-new-plane list-of-values))

(defun init-new-plane (List-of-values)
 (let ((temp (make-plane-name)))
  (set temp (make-instance 'plane
                           :a-coef (rationalize (first list-of-values))
                           :b-coef (rationalize (second list-of-values))
                           :c-coef (rationalize (third list-of-values))
                           :Ao (fourth list-of-values)))
  (push temp *list-of-planes*)
  temp))


;----------------------MAKE ALL FACETS----------------------------
;
; Used by intercept routines to rebuild volume facets.
;
```

```
;
;          ***  W A R N I N G  ***
;
; Note: Facets MUST be convex and MUST NOT be adjacent to
;     facets in the same volume with the same plane equation
;
;---------------------------------------------------------------
; Used by input method 1 and by all intercept routines

(defun make-all-facets (Volume)
 (reset-point-property-lists Volume)
                          ; initialize point 'lines property list
 (loop for L in (Volume-edges (eval Volume))
  do (let* ((endpoints (send (eval L) :endpoints))
            (first-point (first endpoints))
            (second-point (second endpoints)))
          (setf (get first-point 'lines) (adjoin L (get first-point 'lines)))
          (setf (get second-point 'lines) (adjoin L (get second-point 'lines)))))
                          ; build all facets from points
 (loop for P in (volume-points (eval Volume)) ; make all facets possible
  do (loop for L in (get P 'lines)
        do (let* ((other-end-L (send (eval L) :other-end P)))
           (initialize-search Volume P (list L) (List other-end-L P)))))

 (reset-point-property-lists Volume))

(defun initialize-search (Volume  Goal old-lines old-points)
 (let ((point2 (first old-points))
       (Line (first old-lines))
       (search-result 'nil)
       (facet-name 'nil))
  (loop for L in (get point2 'lines)
   do (cond ((and (not (equal L Line))
                  (not (equal Goal (send (eval L) :other-end point2))))
             (let ((plane (init-plane (make-a-normalized-plane L Line))))
              (cond ((not (member-p plane (get Goal 'planes)))
                     (setf (get Goal 'planes) (adjoin plane (get Goal 'planes)))
                     (setf search-result (search-to-make-facet Goal
                                                               plane
                                                               (list L Line)
                                                               (pushnew (send (eval L) :other-end point2)
                                                                        old-points)
                                                               'nil
                                                               'nil))
                     (cond ((<= '3 (length (first search-result)))
                            (setf facet-name (init-facet-2 search-result)))
                           (t (setf facet-name 'nil)))
                     (cond ((not (null facet-name))
                     (setf (volume-facets (eval Volume))
                           (adjoin facet-name (volume-facets (eval Volume))))
                           ))))))))))
```

```
(defun search-to-make-facet (Goal
                             Facet-plane
                             old-lines
                             old-points
                             rejected-points
                             rejected-lines)
 (let ((current-point (first old-points))
       (last-line (first old-lines))
       (Line 'nil)
       (possible-paths 'nil))
   (loop for candidate-line in (get current-point 'lines)
     do (let ((other-end-cand-line
                 (send (eval candidate-line) :other-end current-point)))
       (cond ((apply 'and (list (not (member-p candidate-line old-lines))
                                (not (member-p candidate-line rejected-lines))
                                (not (member-p other-end-cand-line
                                               rejected-points))))
                  (cond ((not (member-p other-end-cand-line old-points))
                         (cond ((send (eval facet-plane) :test-equal
                                      (make-a-plane other-end-cand-line
                                                    (first old-lines)))
                                (setf (get other-end-cand-line 'distance)
                                   (distance Goal other-end-cand-line))
                                (setf possible-paths
                                      (adjoin candidate-line possible-paths)))
                                (t (pushnew candidate-line rejected-lines))))
                        ((equal other-end-cand-line Goal)
                         (loop for P in (adjoin other-end-cand-line old-points)
                           do (setf (get P 'planes)
                                    (adjoin Facet-plane (get P 'planes))))
                         (return-from search-to-make-facet (list
                                                            (adjoin candidate-line
                                                                    old-lines)
                                                            facet-plane)))
                        (t (pushnew candidate-line rejected-lines))))
                 (t (pushnew candidate-line rejected-lines)))))
   (cond ((not (null possible-paths))
          (setf Line (minimum-distance possible-paths current-point))
          (push Line old-lines)
          (pushnew (send (eval Line) :other-end current-point) old-points))
         (t (pushnew last-line rejected-lines)     ; remove last line, current point
           (pushnew current-point rejected-points) ; and retrace steps (backtrack)
           (setf old-lines (rest old-lines))
           (setf old-points (rest old-points))
           (cond ((> 2 (length old-lines))    ; backtracked too far?
                  (return-from search-to-make-facet 'nil)))))
   (search-to-make-facet Goal Facet-plane old-lines old-points
                         rejected-points rejected-lines)))

(defun init-facet-2 (properties)     ; Check to see if already built facet
```

```
 (cond ((not (null properties))      ; else return name of new facet, or nil.
      (let* ((edges (first properties))
             (plane (second properties))
             (test-plane (map 'list 'abs
                              (map 'list '* (send (eval plane) :list-coeff)
                                   *one-vector*)))
             (equal-flag 't))
        (cond ((equal-p test-plane *zero-vector*) ; remove artifact facets
               (return-from init-facet-2 'nil)))
        (cond ((not (null *list-of-facets*))
               (loop for F in *list-of-facets*   ; see if already exists
                 do (cond ((equal (length edges)
                                  (length (facet-edges (eval F))))
                           (setf equal-flag 't)
                           (loop for E in edges
                                 do (cond ((not (member-p E (facet-edges (eval F))))
                                           (setf equal-flag 'nil))))
                           (cond (equal-flag
                                  (return-from init-facet-2 F))))))))
              (make-new-facet edges plane)))
          (t (return-from init-facet-2 'nil))))

(defun make-new-facet (list-of-edges plane)
 (let ((plane-equation (send (eval Plane) :list-coeff))
       (temp (make-facet-name)))
   (set temp (make-instance 'facet
                            :Edges list-of-edges
                            :center 'nil
                            :connects 'nil
                            :a-coef (first Plane-equation)
                            :b-coef (second Plane-equation)
                            :c-coef (third Plane-equation)
                            :Ao (fourth Plane-equation)))
   (push temp *list-of-facets*)
   temp))

;-------------------------MAKE A FACET FROM INPUT----------------------
; Used by input method 2 (only)


(defun make-a-facet (points)   ; build a facet from a list of point names
 (let ((first-point (first points))
       (start-point (first points))
       (lines 'nil)
       (plane-of-facet 'nil))
   (loop for End-point in (rest points)   ; construct edges of facet
     do (let ()
          (setf lines (adjoin (make-line Start-point End-point) lines))
          (setf Start-point End-point)))
   (setf lines (adjoin (make-line Start-point First-point) lines))
```

```
  (setf Plane-of-facet (init-plane (make-a-normalized-plane (first lines)
                                                            (second lines))))
  (make-new-facet lines plane-of-facet)))   ; return new facet name


(defun build-side-facet (Pt1 Pt2 Side-points) ; make a facet w/disjoint list of points
 (make-a-facet (append (list Pt1 Pt2) Side-points)))

(defun build-terrain (data)    ; build facets with raw facet data, where data
                               ; is in format (point point point ...)
                               ; and the points are in format (x y z)
                               ; return a list of all facets built
 (let ((list-of-facets 'nil))
  (loop for Facets in Data     ; each list within data is a facet
   do (let ((points (map 'list #'init-point Facets)))
        (setf list-of-facets (adjoin (make-a-facet points) list-of-facets))))
 list-of-facets))


;------------------------------MAKE A VOLUME---------------------------

(defun init-volume (data)
  (let ((temp (make-volume-name))
        (volume-data (pop data)))
   (set temp (make-instance 'volume
                            :Visibility (second volume-data)
                            :Probability-of-detection 'nil
                            :Composition (first volume-data)
                            :Points '()
                            :Edges '()
                            :Facets '()
                            :arithmetic-center 'nil
                            :connected-to 'nil))
   (push temp *list-of-volumes*)
   (Initialize-volume temp data)
   temp))                        ; return name of volume created
```

FILE NAME: Visibility-functions.lisp

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;;;*****************************************************************
;;;
;;;   V I S I B I L I T Y   A N D   R I D G E S
;;;
;;; This file contains both the visibility determination code
;;; and the ridge creation and initial air-volume "convexizing"
;;; code. The visibility code is first, followed by the ridge
;;; code.
;;;
;;; THESIS                          D.H. Lewis 10/11/86
;;;*****************************************************************
;;;
;;; VISIBILITY REGIONS      D.H. Lewis          10 Aug 88
;;;
;;;          ---------------------------------------
;;;
;;;  Contains the Observer flavor; code for creating and
;;; manipulating observer data; code for making visibility
;;; visibility regions; code for determining the visibility of
;;; visibility volumes; and finally code for finding the probability
;;; of detection for the visibility volumes.
;;;
;;; Main functions:  MAKE-VISIBILITY-REGIONS (OBSERVER)
;;;                  DETERMINE-VISIBILITY (OBSERVER)
;;;                  INIT-OBSERVER (COORDINATES EFFECTIVNESS)
;;;                  CONNECT-VOLUMES ()
;;;
;;; Other functions: MAKE-OBSERVER-NAME
;;;                  COLINEAR-P
;;;                  FIND-T
;;;                  PROBABILITIES-ASSUMING-INDEPENDENCE-OR
;;;                  PROBABILITIES-ASSUMING-INDEPENDENCE-AND
;;;                  CLEAR-VISIBILITY
;;;                  MATCH-FACET-WITH-ANOTHER-VOLUME
;;;                  SHOW-CONNECTIVITY
;;;                  CLEAR-CONNECTIVITY
;;;                  CONNECTIVITY-METRIC
;;;*****************************************************************

(defvar *list-of-observers* 'nil)
(defvar *observer-counter* '0)

;---------------------------------------------------------------
;  FLAVORS USED TO CREATE OR MANIPULATE VISIBILITY REGIONS
;---------------------------------------------------------------

(defflavor Observer
    (Effectivness
```

```
          Position)
     (graphic)     ; for display
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables
  :outside-accessible-instance-variables)

;-------------METHODS FOR OBSERVERS-----------------

(defmethod (observer :make-node-list) ()
 (list (reverse (append (list '1) (reverse (send (eval position) :list-format))))))

(defmethod (observer :make-polygon-list) ()
 '((0 0)))


;----------FUNCTIONS FOR OBSERVERS----------------

(defun make-observer-name ()
 (gensym (incf *observer-counter*))
 (intern (gensym "observer")))


(defun init-observer (coord effectivness)
 (let* ((temp (make-observer-name))
        (position (init-point coord))
        (volume-location (locate-point-air position))) ;which air volumes contain obs?
   (cond ((null volume-location)                ;make sure not underground
          (terpri)
          (princ "Invalid location for observer (underground)") (terpri)
          (return-from init-observer 'nil)))
   (set temp (make-instance 'Observer
                            :Effectivness effectivness
                            :Position position))
   (pushnew temp *list-of-observers*)
   (setf (universe-observers *universe*) (adjoin temp
                                                 (universe-observers *universe*)))
   temp))

;-------------------------------------------------------------
; Determine all observer planes, and make visibility regions
;-------------------------------------------------------------

(defun make-visibility-regions (observer)
 (let ((ground-volumes 'nil)
       (air-volumes 'nil)
       (ridges 'nil)
       (planes 'nil)
       (result-volume-list 'nil))
                  ; find all air,ground volumes, visible ridges
   (terpri) (terpri)
```

```
  (princ "making visibility regions for: ")
  (prin1 observer) (terpri) (terpri)
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes))
             (loop for L in (volume-edges (eval V))
                do (cond ((equal 'ridge (line-segment-characteristics (eval L)))
                          (cond ((not (colinear-p (observer-position (eval observer))
                                                  L))
                                 (setf ridges (adjoin L ridges))))))))
            (t (setf air-volumes (adjoin (list V) air-volumes))
               (setf (universe-volumes *universe*)
                     (remove V (universe-volumes *universe*))))))
                    ; make all visibility limiting planes
  (loop for R in ridges
   do (setf planes (adjoin (make-a-plane (observer-position (eval Observer)) R)
                           planes)))
                    ; intersect all air volumes with planes
  (princ "Air volumes: ") (prin1 air-volumes) (terpri)
  (princ "Limiting planes of visibility: ") (prin1 planes) (terpri) (terpri)
  (setf result-volume-list (intersect-all-planes-with-volumes planes
                                                              air-volumes))
  (loop for V in result-volume-list
   do (push (car V) (universe-volumes *universe*))))
 (send *universe* :save-static-items)      ; save the state of the static model
 (universe-volumes *universe*))

(defun colinear-p (point line)
 (let ((tx (find-t '0 point line))          ; find x,y,z t parameters
       (ty (find-t '1 point line))
       (tz (find-t '2 point line))
       (t-list 'nil)
       (t-list-reduced 'nil))
  (setf t-list (substitute '0.0 'nil (list tx ty tz)))
  (setf t-list-reduced (remove 'nil (list tx ty tz)))
  (cond ((equal '1 (length t-list-reduced))
         (return-from colinear-p
           (apply 'and (mapcar 'equal-error (send (eval point) :list-format-real)
                               (send (eval line) :backsubs t-list)))))
        ((equal '2 (length t-list-reduced))
          (return-from colinear-p (apply 'equal-error t-list-reduced)))
        (t (return-from colinear-p (and (equal-error tx ty)
                                        (equal-error tx tz)))))))

(defun find-t (nr point line)
 (let ((denom (nth nr (send (eval (line-segment-direction-vector
                                   (eval line))) :list-format)))
       (numerator (- (nth nr (send (eval point) :list-format))
                     (nth nr (send (eval (line-segment-position-vector
                                          (eval line))) :list-format)))))
  (cond ((equal-zero-p denom)
```

```
          (return-from find-t 'nil))
         (t (return-from find-t (/ numerator denom))))))


;-----------------------------------------------------------------------
; Determine visibility of visibility regions
;-----------------------------------------------------------------------

(defun determine-visibility (observer)
 (determine-visibility-1 observer))

(defun determine-visibility-1 (observer)
                    ; determine the visibility status (yes or no)
                    ; of all air volumes w/ respect to a sigle observer
                    ; using a fast method
 (terpri) (terpri)
 (princ "Visibility determination for: ") (prin1 observer)
 (terpri) (terpri)
 (let ((ground-volumes 'nil)
       (air-volumes 'nil)
       (ground-facets 'nil)
       (volumes-containing-observer
       (locate-point-air (observer-position (eval observer)))))

                    ; find all air,ground volumes, and ground facets
                    ; make bounding boxes for ground facets

  (set-arithmetic-centers)
  (loop for V in volumes-containing-observer
   do (princ "  ")
   do (prin1 V)
   do (princ " visible")
   do (terpri))
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'air (volume-composition (eval V)))
             (cond ((not (member-p V volumes-containing-observer))
                    (setf air-volumes (adjoin V air-volumes)))))
            (t (setf ground-volumes (adjoin V ground-volumes))
             (loop for F in (volume-facets (eval V))
               do (setf ground-facets (adjoin F ground-facets))))))

                    ; build bounding box for ground facets

  (loop for F in ground-facets
   do (send (eval F) :construct-bounding-box (send (eval F) :points)))

                    ; determine visibility of all air volumes
                    ; containg the observer

  (loop for V in volumes-containing-observer
   do (setf (volume-visibility (eval V))
```

```
                    (adjoin observer (volume-visibility (eval V)))))

                    ; determine visibility of remainder of air volumes
                    ; by seeing if visibility line intersects a ground
                    ; facet

  (loop for V in air-volumes
    do (let ((visibility-line (make-line (observer-position (eval observer))
                                         (volume-arithmetic-center (eval V))))
             (blocked-flag 'nil))
         (loop for F in ground-facets
           do (let ((facet-plane (init-plane (send (eval F) :list-coeff)))
                    (I 'nil))
                (cond ((subs-line-into-plane-equation visibility-line facet-plane))
                      ((not blocked-flag)
                       (cond ((send (eval visibility-line) :strattle-plane-p
                                                           facet-plane)
                              (setf I (find-intercept-point facet-plane
                                                            visibility-line))
                              (cond ((send (eval F) :inside-bounding-box-p I)
                                     (cond ((inside-facet-p I F)
                                            (princ "  ") (prin1 V)
                                            (princ " not visible") (terpri)
                                            (setf blocked-flag 't)))))))))))
         (cond ((not blocked-flag)
                (princ "  ") (prin1 V) (princ " visible") (terpri)
                (setf (volume-visibility (eval V))
                      (adjoin observer (volume-visibility (eval V))))))))
    (terpri)
  'nil))


(defun determine-visibility-2 (observer)
                    ; determine the visibility status (yes or no)
                    ; of all air volumes w/ respect to a sigle observer
                    ; using a slow method
  (terpri) (terpri)
  (princ "Visibility determination for: ") (prin1 observer)
  (terpri) (terpri)
  (let ((ground-volumes 'nil)
        (air-volumes 'nil)
        (ground-facets 'nil)
        (volumes-containing-observer
        (locate-point-air (observer-position (eval observer)))))
    (set-arithmetic-centers)

                    ; determine visibility of all air volumes
                    ; containg the observer

    (loop for V in volumes-containing-observer
      do (setf (volume-visibility (eval V))
```

```
            (adjoin observer (volume-visibility (eval V)))))
  (loop for V in volumes-containing-observer
   do (princ "  ")
   do (prin1 V)
   do (princ " visible")
   do (terpri))

                    ; find who rest of volumes are, and make list
                    ; of blocking ground facets.  Remove all
                    ; vertical ground facets.

  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'air (volume-composition (eval V)))
             (cond ((not (member-p V volumes-containing-observer))
                    (setf air-volumes (adjoin V air-volumes)))))
            (t (setf ground-volumes (adjoin V ground-volumes))
               (loop for F in (volume-facets (eval V))
                  do (cond ((and (member-p '0 (send (eval F) :list-coeff-3))
                                 (> 2 (length (remove '0 (send (eval F)
                                                               :list-coeff-3))))))
                           (t (setf ground-facets (adjoin F ground-facets)))))))))
  (setf ground-facets (remove-duplicates ground-facets))
  (loop for F in ground-facets
   do (send (eval F) :construct-bounding-box (send (eval F) :points)))

                    ; determine visibility of remainder of air volumes
                    ; by seeing if visibility line intersects a ground
                    ; facet

  (loop for V in air-volumes
   do (let ((visibility-line (make-line (observer-position (eval observer))
                                        (volume-arithmetic-center (eval V)))))
        (cond ((find-if-visibility-line-blocked-p visibility-line
                                                  ground-facets
                                                  ground-volumes)
               (princ "  ") (prin1 V)
               (princ " not visible") (terpri))
              (t (princ "  ") (prin1 V) (princ " visible") (terpri)
                 (setf (volume-visibility (eval V))
                       (adjoin observer (volume-visibility (eval
V))))))))
   'nil))

(defun find-if-visibility-line-blocked-p (visibility-line
                                          ground-facets
                                          ground-volumes)
 (loop for F in ground-facets
  do (let ((intercept-point (find-intercept-point
                             (init-plane (send (eval F) :list-coeff))
                             visibility-line))
        (location-volumes 'nil))
```

```
        (cond ((null intercept-point)
                 (return-from find-if-visibility-line-blocked-p 'nil))
               ((not (send (eval F) :inside-bounding-box-p intercept-point))
                 (return-from find-if-visibility-line-blocked-p 't))
               (t (setf location-volumes (locate-point intercept-point))
                 (loop for V in ground-volumes
                   do (cond ((member-p V location-volumes)
                             (return-from find-if-visibility-line-blocked-p 't))))
                 (return-from find-if-visibility-line-blocked-p 'nil))))))


(defun probabilities-assuming-independence-or (volume)
                    ; set volume probability of detection using an
                    ; asssumption of indepedence between observers, and
                    ; an "or" combination technique
 (let ((temp '1.0))
  (terpri)
  (prin1 volume) (princ " has P.D.: ")
  (cond ((not (null (volume-visibility (eval volume))))
         (loop for Obs in (volume-visibility (eval volume))
           do (setf temp (* temp (- '1.0 (observer-effectivness (eval Obs))))))
         (setf (volume-probability-of-detection (eval volume)) (- '1.0 temp))
         (prin1 (- '1.0 temp)))
        (t (setf (volume-probability-of-detection (eval volume)) '0.0)
          (prin1 '0.0)))))

(defun probabilities-assuming-independence-and (volume)
                    ; set volume probability of detection using an
                    ; asssumption of indepedence between observers, and
                    ; an "and" combination technique
 (let ((temp '1.0))
  (terpri)
  (prin1 volume) (princ " has P.D.: ")
  (cond ((not (null (volume-visibility (eval volume))))
         (loop for Obs in (volume-visibility (eval volume))
           do (setf temp (* temp (observer-effectivness (eval Obs)))))
         (setf (volume-probability-of-detection (eval volume)) temp)
         (prin1 temp))
        (t (setf (volume-probability-of-detection (eval volume)) '0.0)
          (prin1 '0.0)))))

(defun set-arithmetic-centers ()
 (loop for V in (universe-volumes *universe*)
  do (setf (volume-arithmetic-center (eval V)) (send (eval V) :find-arithmetic-center))))


(defun clear-visibility ()              ; clear out observer visibility info
 (loop for V in (universe-volumes *universe*)
  do (setf (volume-probability-of-detection (eval V)) 'nil)
  do (setf (volume-visibility (eval V)) 'nil))
 'Done)
```

```
(defun set-zero-PD ()                    ; set all air volume PD's to zero
 (loop for V in (universe-volumes *universe*)
  do (cond ((equal 'air (volume-composition (eval V)))
            (setf (volume-probability-of-detection (eval V)) '0.0))))
 'done)


;*****************************************************************
;
;              C O N N E C T I V I T Y
;
;*****************************************************************


;-------------------------------------------------
;        Connectivity between volumes
;-------------------------------------------------

(defun Connect-volumes ()     ; connect all air volumes by facets.
 (let ((volumes (universe-volumes *universe*)))
  (terpri)
  (terpri) (princ "Connecting volumes:") (terpri) (terpri)
  (loop for V in volumes
   do (prin1 V)
   do (princ " Connected to: ")
   do (setf (volume-connected-to (eval V)) 'nil)
   do (cond ((equal 'air (volume-composition (eval V)))
                  (loop for F in (volume-facets (eval V))
                   do (send (eval F) :find-facet-center)
                   do (send (eval F) :add-volume-to-left-connects V)
                   do (let ((match (match-facet-with-another-volume F V)))
                       (cond ((and
                                  (not (null match))
                                  (not (equal 'ground (volume-composition (eval match)))))
                              (send (eval F) :add-volume-to-right-connects match))
                             ((null match)
                              (let* ((volumes (locate-point-air (facet-center (eval F)))))
                                    (loop for Connect-vol in (remove V volumes)
                                          do (send (eval F) :add-volume-to-right-connects Con-
nect-vol)
                                       ))))))))
   (loop for F in (volume-facets (eval V))
        do (setf (volume-connected-to (eval V))
                 (append (second (facet-connects (eval F)))
                         (volume-connected-to (eval V)))))
   (setf (volume-connected-to (eval V))
         (remove-duplicates (volume-connected-to (eval V))))
   (setf (volume-connected-to (eval V))
         (remove 'nil (volume-connected-to (eval V))))
   (setf (volume-connected-to (eval V))
```

```
            (remove V (volume-connected-to (eval V))))
      (loop for V2 in (volume-connected-to (eval V))        ; remove ground volumes
            do (cond ((equal 'ground (volume-composition (eval V2)))
                      (setf (volume-connected-to (eval V))
                            (remove V2 (volume-connected-to (eval V)))))))
      (prin1 (volume-connected-to (eval V)))
      (terpri))
     (terpri)))

(defun match-facet-with-another-volume (Facet Volume)
                      ; return the name of the unique facet which is shared
                      ; between two volumes, else return NIL. Volume is
                      ; assumed to contain facet
 (let ((volumes (universe-volumes *universe*)))
  (loop for V in volumes
   do (cond ((not (equal V Volume))
                    (cond ((member-p Facet (volume-facets (eval V)))
                           (return-from match-facet-with-another-volume V))
                          ((or (member-p V (second (facet-connects (eval Facet))))
                               (member-p V (first (facet-connects (eval Facet)))))
                           (return-from match-facet-with-another-volume V))))))
  'nil))

(defun show-connectivity ()                    ; show how volumes connect
 (terpri)
 (loop for V in (universe-volumes *universe*)
  do (let ()
       (terpri) (prin1 V)
       (princ " <-> ")
       (prin1 (volume-connected-to (eval V))))))


(defun clear-connectivity ()                   ; clear state of connectivity
 (loop for V in (universe-volumes *universe*)
  do (setf (volume-connected-to (eval V)) 'nil))
 'done)

(defun connectivity-metric ()
 (terpri)
 (loop for V in (universe-volumes *universe*)
  do (prin1 V)
  do (princ ":  Connections: ")
  do (prin1 (length (volume-connected-to (eval V))))
  do (princ "  Facets: ")
  do (prin1 (length (volume-facets (eval V))))
  do (cond ((or (equal (length (volume-connected-to (eval V)))
                       (1- (length (volume-facets (eval V)))))
                (equal (length (volume-connected-to (eval V)))
                       (length (volume-facets (eval V)))))
            (t (princ "  -- possible error")))
  do (terpri)))
```

```
;*****************************************************************
;
;;;
;;;          RIDGE CREATION AND MANIPULATION FUNCTIONS
;;;  D.H. LEWIS                                  22May88
;;;
;;;
;;;          ------------------------------------------
;;;
;;;
;;;  Functions to find, make, and manipulate ridge lines.
;;;
;;;
;;;  Main functions:  FIND-ALL-RIDGES ()
;;;                   LINE-IS-A-RIDGE-P (LINE VOLUME)
;;;                   MAKE-CONVEX-VOLUMES ()
;;;
;;;
;;;  Other functions: FIND-FACETS-WHICH-CONTAIN-EDGE
;;;                   PUT-FACET-ON-CORRECT-SIDE
;;;                   FIND-OVERLAPPING-FACETS
;;;                   FIND-HIGHEST-FACET
;;;                   RIDGE-LENGTH-SORT
;;;
;;;
;*****************************************************************
;

;----Make ridges----

(defun find-all-ridges ()   ; look for line-segments which are ridges
 (terpri) (terpri)
 (princ "Find all ridges in ground terrain: ") (terpri) (terpri)
 (loop for Volume in (universe-volumes *universe*)
  do (cond ((equal 'ground (volume-composition (eval Volume)))
            (loop for E in (Volume-edges (eval Volume))
             do (princ "Ridge check, line: ")
             do (prin1 E)
             do (cond ((line-is-a-ridge-p E Volume)
                       (setf (line-segment-characteristics (eval E))
                             'ridge)
                       (princ " --  Ridge")
                        (terpri))
                      (t (setf (line-segment-characteristics (eval E))
                               'nil)
                         (terpri))))))))

(defun line-is-a-ridge-p (Line Volume)  ; T if line is a ridge
 (let ((Facets (find-facets-which-contain-edge Line Volume))
       (Edge-vertical-plane (make-vertical-plane Line))
       (Right-side-facets 'nil)
       (Highest-right-side-facet 'nil)
       (Left-side-facets 'nil)
       (Highest-left-side-facet 'nil)
       (Vertical-facets 'nil)
       (Overlapping-facets 'nil))
```

```
                ; divide facets into left and right halves based
                ; on spacial relationship of middle point
                ; with vertical plane of Line

(loop for F in facets
  do (setf (get F 'center) (init-point (mean-point-in-facet F )))
  do (setf (get F 'opposite-points) 'nil)
  do (let ((side (put-facet-on-correct-side F Edge-vertical-plane)))
       (cond ((not (null (first side)))
              (setf Left-side-facets (adjoin (first side) Left-side-facets)))
             ((not (null (second side)))
              (setf Vertical-facets (adjoin (second side) Vertical-facets)))
             ((not (null (third side)))
              (setf Right-side-facets (adjoin (third side) Right-side-facets))))))

                ; do not consider vertical facets in any manner

(cond ((not (null Vertical-facets))
       (return-from Line-is-a-ridge-p 'nil)))

                ; handle overlapping facets by creating a new facet center
                ; composed of average of facet points on correct side of
                ; possible ridge line

(cond ((or (null Left-side-facets)
           (null Right-side-facets))
       (cond ((null Left-side-facets)
              (setf Overlapping-facets (find-overlapping-facets Edge-vertical-plane
                                                                Right-side-facets))
              (loop for F in Overlapping-facets
                 do (setf Right-side-facets (remove F Right-side-facets))))
             ((null Right-side-facets)
              (setf Overlapping-facets (find-overlapping-facets Edge-vertical-plane
                                                                Left-side-facets))
              (loop for F in Overlapping-facets
                 do (setf Left-side-facets (remove F Left-side-facets)))))
       (cond ((null Overlapping-facets)          ; have an internal facet
              (return-from line-is-a-ridge-p 'nil)))
       (loop for F in Overlapping-facets
        do (setf (get F 'center) (init-point (average-of-points
                                              (get F 'opposite-points))))
        do (let ((side (put-facet-on-correct-side F Edge-vertical-plane)))
             (cond ((not (null (first side)))
                    (setf Left-side-facets (adjoin (first side) Left-side-facets)))
                   ((not (null (second side)))
                    (setf Vertical-facets (adjoin (second side) Vertical-facets)))
                   ((not (null (third side)))
                    (setf Right-side-facets (adjoin (third side)
                                                    Right-side-facets)))))))))
```

```
                    ; reduce lists of left- and right- facets to one facet
                    ; per side, based upon z-value of mean point of facet

  (cond ((< 1 (length Left-side-facets))
         (setf Highest-left-side-facet (find-highest-facet Left-side-facets)))
        (t (setf Highest-left-side-facet (first Left-side-facets))))
   (cond ((< 1 (length Right-side-facets))
          (setf Highest-right-side-facet (find-highest-facet Right-side-facets)))
         (t (setf Highest-right-side-facet (first Right-side-facets))))

                    ; find if line is a ridge by subs right side mean value
                    ; into left-side plane equation.  If resultant Z value
                    ; is greater than right-side mean value z-value, have
                    ; a ridge, else not

  (let* ((point (send (eval (get Highest-right-side-facet 'center)) :list-format))
       (z-right-point-into-left-plane
    (send (eval Highest-left-side-facet)
                    :find-z-given-xy (first point) (second point))))
    (cond ((> z-right-point-into-left-plane (third point))
           (return-from line-is-a-ridge-p 't))
          (t (return-from line-is-a-ridge-p 'nil))))))



(defun find-facets-which-contain-edge (Edge Volume)
 (let ((temp 'nil))
  (loop for F in (volume-facets (eval Volume))
    do (cond ((member-p Edge (facet-edges (eval F)))
                    (setf temp (adjoin F temp)))))
  temp))

(defun put-facet-on-correct-side (Facet Plane )
 (let* ((Ao (fourth (send (eval plane) :list-coeff)))
       (Ao-Point (subs-point-into-equation (send (eval plane) :list-coeff-3)
                                           (get Facet 'center)))
       (Left 'nil)
       (Vertical 'nil)
       (Right 'nil))
  (cond ((GT Ao Ao-point)
         (pushnew Facet Left))
        ((LT Ao Ao-point)
         (pushnew Facet Right))
        (t (pushnew Facet Vertical)))
  (list (first Left) (first Vertical) (first Right))))


(defun find-overlapping-facets (Vertical-plane Facets)
 (let* ((Line-Ao (fourth (send (eval vertical-plane) :list-coeff)))
       (Facet-center-Ao 'nil)
       (overlapping-facets 'nil))
```

```
  (loop for F in Facets
   do (setf facet-center-Ao (send (eval Vertical-plane) :subs-point-into-plane
                                  (get F 'center)))
   do (loop for P in (send (eval F) :points)
         do (let ((Point-Ao (send (eval Vertical-plane) :subs-point-into-plane P)))
              (cond ((or (and (GT Line-Ao Point-Ao)
                              (LT Line-Ao Facet-center-Ao))
                         (and (LT Line-Ao Point-Ao)
                              (GT Line-Ao Facet-center-Ao)))
                     (setf overlapping-facets (adjoin F overlapping-facets))
                     (setf (get F 'opposite-points)
                           (adjoin P (get F 'opposite-points))))))))
  overlapping-facets))


(defun find-highest-facet (List-of-facets)
 (let ((highest-z (third
                    (send (eval (get (first list-of-facets) 'center)) :list-format)))
       (highest-facet (first List-of-facets)))
  (loop for F in (rest List-of-facets)
       do (let ((z (third (send (eval (get F 'center)) :list-format))))
            (cond ((GT z highest-z)
                   (setf highest-z z)
                   (setf highest-facet F)))))
  highest-facet))



;-------------------------------------------
;---Use ridges to make convex air volumes----
;-------------------------------------------


(defun make-convex-volumes ()      ; intersect all vertical planes from ridge
 (let ((air-volume-list '())       ; line-segments with all volume(s).
       (volume-list 'nil)          ; Makes all air volumes convex,
       (ridge-list 'nil)           ; guarenteed.
       (plane-list 'nil))

  (terpri) (terpri)
  (princ "Making air volumes convex:")
  (terpri) (terpri)

                  ; seperate all air and ground volumes
                  ; and find ridge lines

  (loop for V in (Universe-volumes *universe*)
        do (cond ((equal 'air (volume-composition (eval V)))
                  (setf air-volume-list (adjoin (list V) air-volume-list))
                  (loop for E in (volume-edges (eval V))
```

```
                    do (cond ((equal 'ridge (line-segment-characteristics (eval E)))
                             (setf ridge-list (adjoin E ridge-list)))))
                   (setf (universe-volumes *universe*)
                      (remove V (universe-volumes *universe*))))))

                  ; reduce list of ridge lines, and construct vertical planes
                  ; for them.  ridges are sorted by length, longest first

  (setf ridge-list (remove-duplicates ridge-list))
  (setf ridge-list (remove 'nil ridge-list))
  (setf ridge-list (stable-sort ridge-list #'ridge-length-sort))
  (loop for R in ridge-list
    do (setf plane-list (adjoin (make-vertical-plane R) plane-list)))
  (setf plane-list (reverse plane-list))
  (princ "Air volumes: ") (prin1 air-volume-list) (terpri)
  (princ "Ridge planes: ") (prin1 plane-list) (terpri) (terpri)

                  ; intersect all ridge planes with all air volumes

  (setf volume-list (intersect-all-planes-with-volumes plane-list
                                                       air-volume-list))

                  ; update universe with new volumes created

  (loop for V in volume-list
    do (push (car V) (universe-volumes *universe*)))
  (universe-volumes *universe*)))


(defun ridge-length-sort (A B)          ;return T iff A > B
  (> (send (eval A) :length)
    (send (eval B) :length)))
```

FILE NAME: Path-planning.lisp

```
;;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************************
;
;
; PATH PLANNING              D.H. Lewis              25 Aug 88
;      ------------------------------------------------------
;
; Contains the flavors, methods, and functions nessesary to conduct path
; planning. Divided into four main sections; Flavors, A-star
; path planning, and path optimization.
; The flavors section provides the essential path and agenda item flavors,
; and their associated method and support functions.
; The A* search section conducts an a* search of the volumes, minimizing
; cost and visibility, and creates an initial path.
; Finally, the optimization code optimizes the initial A* path according
; to snells law criteria. This section may create one or several paths
;
;*****************************************************************************

(defvar *PD-threshold* '0.0)        ; maximum desirable probability of detection
(defvar *PD-modifier* '10.0)        ; affects effect of PD on path planning
(defvar *PI* '3.14159)

(defvar *path-counter* '0)          ; path name variables
(defvar *list-of-paths* 'nil)       ; location of all instanciated paths
(defvar *agenda-counter* '0)        ; agenda instanciations

(defvar *Turn45* '10.0)             ; cost for turn of 45 degrees or less
(defvar *Turn90* '50.0)             ; cost for turn between 45 and 90 degrees
(defvar *BigTurn* '5000.0)          ; cost for turns greater than 90 degrees

(defvar *Shallow-Climb* '1.2    ; ratio modifier for a shallow climb
(defvar *Steep-Climb* '1.80)    ; ratio modifier for a steep climb
(defvar *Dive* '0.80)           ; ratio modifier for any dive

;*****************************************************************************
;
;      FLAVORS, METHODS, AND FUNCTIONS
;
;*****************************************************************************

;-------------------------------------------------
;              PATH FLAVOR
;-------------------------------------------------

(defflavor path
     (start-point
          end-point              ; goal
          volumes                ; general path "corridor"
          facets                 ; "windows" in cooridor
```

```
          lines                ; specific path to follow
          points               ; turn points in path
          length               ; of current lines
          total-K              ; sum of deviations from snells law for path
          max-detection-probability
          ave-detection-probability)  ; average of entire path corridor
         (graphic)
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables
 :outside-accessible-instance-variables)

;----------METHODS FOR PATHS----------

(defmethod (path :length) ()    ; find the total length of the path
 (let ((val '0.0))
  (cond ((null length)
         (loop for L in lines
           do (setf val (+ val (send (eval L) :length))))
         (setf length val)))
  length))

(defmethod (path :max-detection-probability) ()  ; find the highest PD on the path
 (let ((maximum (volume-probability-of-detection (eval (first volumes)))))
  (loop for V in (rest volumes)
    do (cond ((< maximum (volume-probability-of-detection (eval V)))
              (setf maximum (volume-probability-of-detection (eval V))))))
  (setf max-detection-probability maximum)))

(defmethod (path :ave-detection-probability) ()  ; find the weighted average of the PD's
 (let ((weighted-sum '0.0))
  (loop for Counter from 0 to (1- (length volumes))
    do (setf weighted-sum
             (+ weighted-sum
                (* (send (eval (nth Counter lines)) :length)
                   (volume-probability-of-detection (eval (nth Counter Volumes)))))))
  (setf ave-detection-probability (/ weighted-sum
                                     (send self :length)))
 ave-detection-probability))

(defmethod (path : make-node-list) ()     ; used by graphic mixin-flavor to draw
 (loop for P in points
  collect (reverse (append (list '1) (reverse (send (eval P) :list-format))))))



(defmethod (path :make-polygon-list) ()   ; used by graphic mixin-flavor to draw
 (loop for L in lines
    do (setf Pt1 (car (send (eval L) :endpoint-list)))
    do (setf Pt2 (cadr (send (eval L) :endpoint-list)))
     collect (list (position-if '(lambda (A) (equal A Pt1)) node-list)
```

```
                    (position-if '(lambda (A) (equal A Pt2)) node-list))))

;----------------------PATH NAMES----------------------


(defun make-path-name ()        ; make a new name for a path
 (gensym (incf *path-counter*))
 (intern (gensym "path")))

(defun init-new-path (start end volumes facets lines points length K) ;make a new path
 (let ((name (make-path-name)))
  (set name (make-instance 'path
                           :start-point start
                           :end-point end
                           :volumes volumes
                           :facets facets
                           :lines lines
                           :points points
                           :length length
                           :total-K K
                           :max-detection-probability 'nil
                           :ave-detection-probability 'nil))
  (push name *list-of-paths*)
  name))



;-----------------------------------------------------
;              AGENDA-ITEM FLAVOR
;-----------------------------------------------------

(defflavor agenda-item
          (volume
           cost
           evaluation
           path)
          ()
 :gettable-instance-variables
 :settable-instance-variables
 :inittable-instance-variables
 :outside-accessible-instance-variables)



;------------------AGENDA-ITEM NAMES------------------

(defun make-agenda-item-name ()
 (gensym (incf *agenda-counter*))
 (intern (gensym "agenda")))

(defun init-agenda-item (volume cost evaluation path)
```

```
(let ((name (make-agenda-item-name)))
  (set name (make-instance 'agenda-item
                           :volume volume
                           :cost cost
                           :evaluation evaluation
                           :path path))
  name))


;*****************************************************************************
;
;
;    S   E   A   R   C   H   E   S
;
;
;*****************************************************************************


;-------------------------------------------------
;              A* Search
;-------------------------------------------------

(defun A-star-search (Start-point End-point Trace-flag)
 (let* ((start-volume (first (locate-point-air start-point)))
        (goal-volume (first (locate-point-air end-point)))
        (successor-volumes (volume-connected-to (eval start-volume)))
        (path-volumes 'nil)
        (agenda 'nil)
        (best-path))

   (terpri) (terpri)
   (princ ">>>>Begin A-star Search") (terpri) (terpri)
   (princ "     Start Volume: ") (prin1 start-volume) (terpri)
   (princ "     Goal Volume: ") (prin1 goal-volume) (terpri) (terpri)
   (cond (trace-flag
            (terpri) (princ "Search trace selected.  Top five and bottom five items")
            (terpri) (princ "on seach agenda will be printed.") (terpri) (terpri))
          (t (terpri)))
   (princ "Search")

                  ; initalize the search agenda

   (setf agenda (put-successors-on-agenda
                                start-volume        ; end of last path
                                successor-volumes   ; successors to be added
                                '0.0                ; cost
                                (list start-volume) ; path to date
                                end-point           ; goal
                                agenda))            ; agenda to be changed

                  ; SEARCH along best agenda item for all possible paths
                  ; until get to the goal along one of the paths
```

```
(loop until (goal-on-agenda-p goal-volume agenda)
 do (princ ".")
 do (cond (trace-flag
                (princ "----------New Agenda----------")
          (print-agenda agenda)))
 do (let* ((best-successor-volume (first agenda))
           (successors-to-best (volume-connected-to (eval (agenda-item-volume
                                                           (eval best-successor-volume))))))
          (setf successors-to-best (remove 'EDGE successors-to-best))
          (loop for V in (rest (agenda-item-path (eval best-successor-volume)))
           do (setf successors-to-best (remove V successors-to-best)))
          (setf agenda (remove best-successor-volume agenda))
          (setf agenda (put-successors-on-agenda
                                  (agenda-item-volume (eval best-successor-volume))
                                     successors-to-best
                                     (agenda-item-cost (eval best-successor-volume))
                                     (agenda-item-path (eval best-successor-volume))
                                     end-point
                                     agenda))))

                ; SEARCH COMPLETED!

                    ; find lines and points in search

(setf path-volumes (reverse (find-path goal-volume agenda))) ;get resultant path
(setf best-path (init-new-path start-point
                               end-point
                               path-volumes
                               'nil
                               'nil
                               'nil
                               'nil
                               'nil))
(princ "Completed") (terpri) (terpri)
(make-facet-to-facet-path best-path)      ; make initial guess at optimal path
(calc-path-and-stats best-path)          ; fill out rest of path flavor data
best-path))



;-----------------------------------------------------
;    A* Search with multiple solutions
;-----------------------------------------------------

(defun A-star-search-M (Start-point End-point Trace-flag paths)
 (let* ((start-volume (first (locate-point-air start-point)))
        (goal-volume (first (locate-point-air end-point)))
        (successor-volumes (volume-connected-to (eval start-volume)))
        (path-volumes 'nil)
        (agenda 'nil)
```

```
    (paths-found))

(terpri)
(princ ">>>>Begin A-star Search") (terpri) (terpri)
(princ "    Start Volume: ") (prin1 start-volume) (terpri)
(princ "    Goal Volume: ") (prin1 goal-volume) (terpri) (terpri)
(cond (trace-flag
        (terpri) (princ "Search trace selected.  Top five and bottom five items")
        (terpri) (princ "on seach agenda will be printed.") (terpri) (terpri))
      (t (terpri)))

            ; initalize the search agenda

(setf agenda (put-successors-on-agenda
                          start-volume        ; end of last path
                          successor-volumes   ; successors to be added
                          '0.0                ; cost
                          (list start-volume) ; path to date
                          end-point           ; goal
                          agenda))            ; agenda to be changed

            ; SEARCH along best agenda item for all possible paths
            ; until get to the goal along one of the paths

(loop repeat paths    ; find top several paths
 do (terpri)
 do (princ "Search")
 do (loop until (goal-on-agenda-p goal-volume agenda) ; same loop as single search
    do (princ ".")
    do (cond (trace-flag
                (princ "----------New Agenda----------")
        (print-agenda agenda)))
    do (let* ((best-successor-volume (first agenda))
              (successors-to-best (volume-connected-to (eval (agenda-item-volume
                                                    (eval best-successor-volume))))))
        (setf successors-to-best (remove 'EDGE successors-to-best))
        (loop for V in (rest
                                (agenda-item-path (eval best-successor-volume)))
          do (setf successors-to-best (remove V successors-to-best)))
        (setf agenda (remove best-successor-volume agenda))
        (setf agenda (put-successors-on-agenda
                                (agenda-item-volume (eval best-successor-volume))
                                   successors-to-best
                                   (agenda-item-cost (eval best-successor-volume))
                                   (agenda-item-path (eval best-successor-volume))
                                   end-point
                                   agenda))))
        (setf path-volumes (reverse (find-path goal-volume agenda)))
        (setf agenda (remove-goal goal-volume agenda))
        (setf paths-found (adjoin (init-new-path start-point
                                                 end-point
```

```
                                                    path-volumes
                                                    'nil
                                                    'nil
                                                    'nil
                                                    'nil
                                                    'nil)
                                        paths-found))

            (princ "Completed") (terpri) (terpri)
                    (make-facet-to-facet-path (first paths-found))
            (calc-path-and-stats (first paths-found)))
  paths-found))


;-----------------------------------------------
;       Search utility functions
;-----------------------------------------------


;-------agenda manipulations-------

                        ; for A-STAR search
(defun put-successors-on-agenda (pred-volume
                                 successor-volumes
                                 cost           ; cost so far
                                 path           ; volumes
                                 goal
                                 agenda)
 (loop for V in successor-volumes
  do (setf agenda (adjoin (init-agenda-item V           ; name
                                            (+ cost (cost-function V path))
                                            (evaluation-function pred-volume
                                                                 V
                                                                 path
                                                                 goal)
                                            (adjoin V path))   ; path
                          agenda)))
 (stable-sort agenda #'agenda-sort-p))

(defun agenda-sort-p (A B)
 (cond ((LT (+ (agenda-item-cost (eval A))
               (agenda-item-evaluation (eval A)))
            (+ (agenda-item-cost (eval B))
               (agenda-item-evaluation (eval B))))
        (return-from agenda-sort-p 't)))
 'nil)

(defun goal-on-agenda-p (goal agenda)   ; return T iff goal volume is on the agenda
 (loop for A in agenda
  do (cond ((equal goal (agenda-item-volume (eval A)))
            (return-from goal-on-agenda-p 'T))))
```

```
 'nil)

(defun remove-goal (goal agenda)
 (loop for A in agenda
  do (cond ((equal goal (agenda-item-volume (eval A)))
            (return-from remove-goal (remove A agenda)))))
 'nil)

(defun find-path (goal agenda)        ; get the path once the goal is found
 (loop for A in agenda
  do (cond ((equal goal (agenda-item-volume (eval A)))
            (return-from find-path (agenda-item-path (eval A)))))))

(defun print-agenda (agenda)          ; print agenda and some/all items on the agenda
 (terpri) (pprint agenda) (terpri)
 (cond ((>= 10 (length agenda))
        (princ "Entire agenda: ") (terpri)    ; print whole agenda if short
         (loop for I in agenda
          do (terpri)
          do (describe I)))
       (t (princ "First five in agenda: ") (terpri) ; do first five and last five
         (loop for Count in '(0 1 2 3 4)          ; if long
          do (describe (nth count agenda))
          do (terpri))
         (terpri) (princ "Last five on agenda: ") (terpri)
         (loop for Count in '(6 5 4 3 2 1)
          do (describe (nth (- Count (length agenda)) agenda))
          do (terpri))))
  (terpri) (terpri))


;-----evaluation and cost functions------

(defun evaluation-function (VP VS path-volumes Goal)
 (let ((turn-modifier (eval-turn-cost VP VS path-volumes))
       (altitude-modifier (eval-climb-dive VP VS))
       (PD-modifier (+ '1.0 (* *PD-modifier*
                               (- (volume-probability-of-detection (eval VS))
                                  *PD-threshold*))))
      (basis-distance (distance (volume-arithmetic-center (eval VS)) Goal)))
  (* PD-modifier (* altitude-modifier (+ turn-modifier basis-distance)))))

(defun cost-function (VS path-volumes)
 (let ((altitude-modifier (eval-climb-dive (first path-volumes) VS))
       (turn-modifier (eval-turn-cost (first path-volumes) VS path-volumes))
       (PD-modifier '0.0)
       (basis-cost (distance (volume-arithmetic-center (eval VS))
                             (volume-arithmetic-center (eval (first path-volumes))))))
  (loop for V in path-volumes
   do (setf PD-modifier (+ PD-modifier
                           (volume-probability-of-detection (eval V)))))
```

```
  (setf PD-modifier (/ PD-modifier (length path-volumes)))
  (setf PD-modifier (+ '1.0 (* *PD-modifier*
                               (- PD-modifier *PD-threshold*))))
  (* PD-modifier (* altitude-modifier (+ turn-modifier basis-cost)))))


(defun eval-turn-cost (VP VS Path-volumes)
 (let ((projected-VP-center (project-xy (volume-arithmetic-center (eval VP))))
       (projected-VS-center (project-xy (volume-arithmetic-center (eval VS))))
       (previous-volume (find-previous-volume VP Path-volumes))
       (projected-vol-center 'nil)
        (path 'nil)
        (new-path 'nil)
        (angle-of-turn 'nil))

  (cond ((equal VP previous-volume)            ; no previous path ?
           (return-from eval-turn-cost '1.0))
         (t (setf projected-vol-center (project-xy
                                  (volume-arithmetic-center (eval previous-volume))))
           (setf path (make-line projected-vol-center projected-VP-center))
           (setf new-path (make-line projected-VP-center projected-VS-center))
           (setf angle-of-turn (angle-between-lines path new-path))
           (cond ((null angle-of-turn)
                   (return-from eval-turn-cost '0.0)))       ; no turn
            (cond ((GT (/ *PI* '4.0) angle-of-turn)
                   (return-from eval-turn-cost *Turn45*)))   ; turn < 45 degrees
           (cond ((GT (/ *PI* '2.0) angle-of-turn)
                   (return-from eval-turn-cost *Turn90*)))))  ; 90 < turn < 45
  *BigTurn*))                                  ; turn > 90


(defun project-xy (Point)
 (let ((point-coords (send (eval Point) :list-format)))
  (init-point (list (first point-coords) (second point-coords) '0.0))))

(defun find-previous-volume (VP path-volume)
 (let ((position-VP (position VP path-volume)))
  (cond ((> 1 (length path-volume))
          (return-from find-previous-volume (elt (1+ position-VP) path-volume)))
         (t (return-from find-previous-volume (first path-volume))))))

(defun eval-climb-dive (VP VS)
 (let* ((inter-facet (find-common-facet VP VS))
       (interfacet-z (third (mean-point-in-facet inter-facet)))
       (path-z (third
                  (send (eval (volume-arithmetic-center (eval VP))) :list-format))))
  (cond ((and (LT path-z (* interfacet-z '1.10))
              (GT path-z (* interfacet-z '0.90)))
          (return-from eval-climb-dive '1.0))          ; level flight
         ((GT path-z interfacet-z)
          (return-from eval-climb-dive *Dive*))        ; dive
```

```
        (t (loop for P in (send (eval inter-facet) :points)
             do (cond ((> path-z (third (send (eval P) :list-format)))
                                                  ; shallow climb
                       (return-from eval-climb-dive *Shallow-Climb*))))))
  *Steep-Climb*))                                 ; steep climb


;-----general functions in support of path planning------


(defun Calc-path-and-stats (path)        ; used to find support info on a new path
 (send (eval path) :length)

                    ; determine probability limits

 (send (eval path) :max-detection-probability)
 (send (eval path) :ave-detection-probability)
 (princ ">>>>Path Statistics:") (terpri) (terpri)
 (princ "  Maximum detection probability: ")
 (prin1 (path-max-detection-probability (eval path)))
 (terpri)
 (princ "  Average detection probability: ")
 (prin1 (path-ave-detection-probability (eval path)))
 (terpri)
 (princ "  Total length of path: ")
 (prin1 (path-length (eval path)))
 (terpri)
 (princ "  Total number of maneuvers: ") (prin1
                                          (- (length (path-points (eval path))) '2))
 (terpri) (terpri)
 (princ ">>>>Path: ") (prin1 path) (terpri) (terpri)

 'nil)

(defun find-intermediate-facets (path)          ; find all the facets along
                                  ; the path
 (let ((previous-volume (first (path-volumes (eval path))))
       (facets 'nil))
  (loop for V in (rest (path-volumes (eval path)))
   do (setf facets (adjoin (find-common-facet previous-volume V) facets))
   do (setf previous-volume V))
  (reverse facets)))

(defun make-facet-to-facet-path (path)
 (let ((last-point (path-start-point (eval path)))
       (points (path-start-point (eval path)))
       (lines 'nil))
  (setf (path-facets (eval path)) (find-intermediate-facets path))
  (loop for F in (path-facets (eval path))
   do (let ((next-point (init-point (mean-point-in-facet F))))
        (setf lines (adjoin (make-line last-point next-point) lines))
```

```
            (setf points (adjoin next-point points))
            (setf last-point next-point)))
    (push (make-line last-point (path-end-point (eval path))) lines)
    (push (path-end-point (eval path)) points)
    (setf (path-lines (eval path)) (reverse lines))
    (setf (path-points (eval path)) (adjoin (path-start-point (eval path)) (reverse points)))))



;*****************************************************************************
;
;              PATH  OPTIMIZATION
;
;*****************************************************************************
;;;
;;; OPTIMIZE PATH ACCORDING TO SNELL'S LAW.          D.H. LEWIS  10/11/88
;;;
;;;  Develop an expression for snell's constant at each facet along the
;;; the path, and then minimize it with respect to the facets before
;;; and after the facet concerned.  Sum all constants along the path
;;; to determine the net amount of deviation from snell's law. Repeat
;;; until total constant minimized.
;;;
;*****************************************************************************

(defvar *PI2* (/ *PI* '2.0))
(defvar *search-increment* '10)

;--------------------MAIN PATH OPTIMIZATION FUNCTION--------------------

(defun optimize-path (path)
 (let ((new-path-points (list (path-start-point (eval path))))
       (new-path-lines 'nil)
       (new-path-length '0.0)
       (last-point 'nil)        ; dummy for building path lines
       (total-K '0.0))          ; total deviation from snell's law

                        ; optimize path point for each facet in turn,
                        ; appending new points onto new-point list as
                        ; they are created

   (terpri) (terpri)
   (princ "Optimizing path ") (prin1 path) (princ ":") (terpri) (terpri)
   (loop for Facet-nr from '1 to (length (path-facets (eval path)))
    do (let ()
         (princ "Optimizing at facet number ")
         (prin1 facet-nr) (princ " : ") (prin1 (nth (1- facet-nr) (path-facets (eval path))))
         (terpri))
    do (let ((prev-point 'nil)
             (next-point (nth (1+ facet-nr) (path-points (eval path))))
             (path-point (nth facet-nr (path-points (eval path)))
```

```
            (new-point 'nil)
            (facet (nth (1- facet-nr) (path-facets (eval path))))
            (N1 (+ '1 (volume-probability-of-detection
                              (eval (nth (1-  facet-nr) (path-volumes (eval path)))))))
            (N2 (+ '1 (volume-probability-of-detection
                              (eval (nth facet-nr (path-volumes (eval path))))))))

                              ; use "best" previous point estimate

          (cond ((> facet-nr '1)
                  (setf prev-point (first new-path-points)))
                 (t (setf prev-point (nth (1- facet-nr) (path-points (eval path))))))

;         (pprint (list '"initial: " facet-nr prev-point path-point next-point facet N1 N2))
          (setf new-point (optimize-point-on-facet prev-point
                                                   next-point
                                                   facet
                                                   path-point
                                                   N1
                                                   N2))
;         (pprint (list '"new path point: " new-point (get new-point 'K)))
          (setf new-path-points (adjoin new-point new-path-points))
          (setf total-K (+ total-K (get new-point 'K)))))

                              ; add goal to new points, draw new path

    (setf new-path-points (adjoin (car (last (path-points (eval Path)))) new-path-points))
    (setf new-path-points (reverse new-path-points))
    (setf last-point (first new-path-points))
    (loop for P in (rest new-path-points)
      do (let ()
           (setf new-path-lines (adjoin (make-line last-point P) new-path-lines))
           (setf new-path-length (+ (send (eval (first new-path-lines)) :length)
                                    new-path-length))
           (setf last-point P)))
    (setf new-path-lines (reverse new-path-lines))

                              ; build the new path with optimized path data

    (terpri) (terpri)
    (princ "Optimization completed") (terpri)
    (calc-path-and-stats (init-new-path (path-start-point (eval path))
                                        (path-end-point (eval path))
                                        (path-volumes (eval path))
                                        (path-facets (eval path))
                                        new-path-lines
                                        new-path-points
                                        new-path-length
                                        total-K))))
```

```
;----------------FIND THE BEST POINT ON THE FACET----------------


(defun optimize-point-on-facet (prev-point next-point facet path-point N1 N2)

                        ; Find the point on the facet with the lowest
                        ; snell's constant (K).

 (let* ((straight-path-line (make-line prev-point next-point))
        (straight-path-point (find-intercept-point facet straight-path-line))
        (path-K-line (make-line path-point straight-path-point))
        (path-plane (make-a-plane prev-point path-K-line))
        (list-of-points (find-edge-points-of-facet path-plane facet)))
;   (pprint list-of-points)
;   (pprint (list facet straight-path-point))
   (setf (get straight-path-point 'K) (find-snells-constant
                                              straight-path-point
                                              (make-line straight-path-point  prev-point)
                                              (make-line straight-path-point  next-point)
                                              facet
                                              N1
                                              N2))

                        ; do special cases first

   (cond ((inside-facet-p straight-path-point facet)
            (cond ((equal '0.0 (* '1.0 (get straight-path-point 'K)))
                     (return-from optimize-point-on-facet straight-path-point))
                  (t (setf list-of-points (adjoin straight-path-point list-of-points)))))
          (t (setf list-of-points (adjoin path-point list-of-points))))
;  (pprint (list list-of-points (length list-of-points)))
   (cond ((< '1 (length list-of-points))
            (setf path-point (optimize-K-on-line list-of-points
                                              prev-point
                                              next-point
                                              facet
                                              N1
                                              N2)))
          (t (setf (get path-point 'K) (find-snells-constant Path-point
                                              (make-line Path-point prev-point)
                                              (make-line Path-point next-point)
                                              facet
                                              N1
                                              N2))))
   path-point))


(defun optimize-K-on-line (agenda prev-point next-point facet N1 N2)
  (let ((lowest-K-point 'nil)
```

```
        (best-line 'nil)
        (mid-point 'nil))
;   (pprint (list '"Optimize: " agenda))
   (loop for P in agenda
    do (setf (get P 'K) (find-snells-constant P
                                              (make-line P prev-point)
                                              (make-line P next-point)
                                              facet
                                              N1
                                              N2)))
   (setf agenda (stable-sort agenda #'agenda-sort-on-K))
   (setf lowest-K-point (first agenda))
;  (pprint (list '"Sorted optimize: " agenda lowest-K-point))

   (loop repeat '3
    do (let ()
         (setf best-line (make-line (first agenda) (second agenda)))
         (setf mid-point (init-point (send (eval best-line) :midpoint)))
         (setf (get mid-point 'K) (find-snells-constant mid-point
                                              (make-line mid-point prev-point)
                                              (make-line mid-point next-point)
                                              facet
                                              N1
                                              N2))
         (setf agenda
               (stable-sort (list (first agenda) (second agenda) mid-point)
                            #'agenda-sort-on-K))
;         (pprint agenda)
;         (pprint (list (first agenda) (get (first agenda) 'K)))
         ))

   (first agenda)))


(defun find-edge-points-of-facet (plane facet)
 (let ((intercept-points 'nil))
  (loop for E in (facet-edges (eval facet))
   do (let ((intercept-point (find-intercept-point plane E)))
        (cond ((not (null intercept-point))
               (setf intercept-points (adjoin intercept-point intercept-points))))))
 intercept-points))


(defun agenda-sort-on-K (A B)   ; sort by increasing absolute value of K property
 (< (abs (get A 'K)) (abs (get B 'K))))


;----------FIND SNELLS CONSTANT----------
```

```
(defun find-snells-constant (Point Line-1 Line-2 Facet N1 N2)
                            ; find snells constant at a boundary, i.e.:
                            ;
                            ;       K = N1 * sin(theta-1) - N2 * sin(theta-2)
                            ;
                            ; note: returns NIL if anything would "blow this up"

 (let* ((end-point-normal-line
          (init-point (map 'list '+ (send (eval Point) :list-format)
                                    (map 'list '* '(100 100 100)
                                    (send (eval facet) :list-coeff-3)))))
         (normal-line (make-line Point end-point-normal-line))
         (perpendicular-plane
          (make-a-plane
           (init-point (list '0 '0 (third (send (eval point) :list-format))))
           normal-line))
         (line-joining-points (make-line (send (eval line-1) :end-point)
                                         (send (eval line-2) :end-point)))
         (default '100)
         (theta-1 (angle-between-lines Line-1 normal-line))
         (theta-2 (angle-between-lines Line-2 normal-line)))
  (cond ((and (not (null Theta-1))
              (not (null theta-2)))
         (setf theta-1 (abs (realpart theta-1)))
         (setf theta-2 (abs (realpart theta-2)))
         (cond ((< *PI2* theta-1)
                (setf theta-1 (- *PI* theta-1))))
         (cond ((< *PI2* theta-2)
                (setf theta-2 (- *PI* theta-2))))
         (cond ((> theta-1 (realpart (asin (/ N2 N1))))       ; critical angle?
                (setf theta-2 *PI2*)))
         (cond ((send (eval line-joining-points) :strattle-plane-p perpendicular-plane)
                (return-from
                        find-snells-constant (- (* N1 (sin theta-1))
                                                (* N2 (sin theta-2)))))
               (t (return-from
                       find-snells-constant (- (* N1 (sin theta-1))
                                               (* N2 (- (* '2 *PI*)
                                                        (sin theta-1)))))))))
  default))
```

FILE NAME: Common-functions.lisp

```
;; -*- Mode:Common-Lisp; Base:10 -*-
;*****************************************************************************
;
;
;          COMMON FUNCTIONS
;
;        ----------------------------------------
;
; This file consists of all common functions used by most of the
; files of the 3-D path planning software. Function vary from the
; very general (convenience) functions, to very detailed, special
; purpose functions (which happen to be called from two seperate
; files). Functions are grouped by categories of Simple functions,
; Point functions, Vector functions, Line functions, Plane functions,
; Facet functions, Volume functions, property list functions,
; detailed (special purpose) functions, and finally, printing functions.
;
; D.H.Lewis/Thesis                          07 AUG 88
;*****************************************************************************
;
;        DIRECTORY OF FUNCTIONS
;
;     ------------------------------------------------------------
;
; SIMPLE: MEMBER-P          POINTS: AVERAGE-OF-POINTS
;       EQUAL-P                     FIND-POINT
;       EQUAL-ZERO-P                AVERAGE-POINT
;       DISTANCE
;       MERGE-JOIN-LIST
;       FIRST-NON-ZERO      VECTORS: SOLVE-FOR-T
;       EQUAL-ERROR                  VECTOR-ADD-WITH-T
;       LT, GT, GE, LE
;
; LINES:  MAKE-LINE         PLANES:  MAKE-A-PLANE
;       LINE-CROSS-PRODUCT           MAKE-A-NORMALIZED-PLANE
;       FIND-COMMON-POINT            MAKE-VERTICAL-PLANE
;       ANGLE-BETWEEN-LINES          MAKE-Z-PLANE
;                                   MAKE-X-PLANE
; FACETS:  FIND-COMMON-FACET        MAKE-Y-PLANE
;       MEAN-POINT-IN-A-FACET       SUBS-POINT-INTO-EQUATION
;       MEAN-POINT-IN-A-FACET-2     SUBS-LINE-INTO-PLANE-EQUATION
;       INFO-ON-FACETS
;       INSIDE-FACET-P
;
; VOLUMES: INTERSECT-ALL-PLANES-WITH-VOLUMES
;
; PROPERTY LISTS: RESET-POINT-PROPERTY-LISTS
;
; DETAILED FUNCTIONS: MINIMUM-DISTANCE
```

```
;               LOCATE-POINT-AIR
;               POINT-IN-VOLUME-P
;               POINT-CHECK-P
;               LINES-STRATTLE-FACETS-P
;               SPEED-DEMON
;
; PRINTING FUNCTIONS: DUMP-VOLUMES
;               DDUMP-PATH
;               PRINT-POINTS
;               PRINT-VECTORS
;               PRINT-LINES
;               PRINT-FACETS
;               PRINT-VOLUMES
;
;*****************************************************************************

(defvar *precision* '0.0025)

;-----------------SIMPLE FUNCTIONS---------------


(defun member-p (item list)                    ; T or nil member
  (not (null (member item list))))

(defun equal-p (list1 list2)                   ; are two lists equal?
  (cond ((equal (length list1) (length list2))
         (apply 'and (mapcar 'equal list1 list2)))))

(defun equal-zero-p (A)                        ; is A equal to zero?
  (cond ((equal (* '1.0 A) '0.0)
             (return-from equal-zero-p 't)))
  'nil)

(defun distance (P1 P2)                        ; distance between two points
  (let ((difference (mapcar '- (send (eval P1) :list-format)
                               (send (eval P2) :list-format))))
    (sqrt (apply '+ (mapcar '* difference difference)))))

(defun merge-join-list (List1 List2)           ; join the two lists to make
  (let ((length1 (length list1))               ; one long list
        (length2 (length list2))
        (templist 'nil))
    (cond ((>= length1 length2)
           (setf templist list1)
           (loop for I in list2
             do (setf templist (adjoin I templist))))
          (t (setf templist list2)
             (loop for I in list1
               do (setf templist (adjoin I templist)))))
    templist))
```

```
(defun first-non-zero (List)      ; find the first non-zero element in a simple list
                                  ; if none found, return "-1".
 (cond ((not (equal-zero-p (first List)))
        (first List))
       ((not (equal-zero-p (second List)))
        (second List))
       ((not (equal-zero-p (third List)))
        (third List))
       (t (- 1))))

(defun equal-error (A B)                ; equal within an allowed level of error
 (let ((error 'nil))
  (cond ((equal A B)                    ; simple equal
         (return-from equal-error 't))
        ((equal (* '1.0 A)              ; floating point equal
                (* '1.0 B))
         (return-from equal-error 't))
        ((or (equal-zero-p B)           ; divide by zero check
             (equal-zero-p A))
         (setf error '1.0))
        ((> A B)                        ; find absolute error between terms
         (setf error (abs (/ (- A B) B))))
        (t (setf error (abs (/ (- A B) A)))))
  (<= error *precision*)))              ; check with allowed precision

(defun LT (A B)
 (and (not (equal-error A B))
      (< A B)))

(defun GT (A B)
 (and (not (equal-error A B))
      (> A B)))

(defun LE (A B)
 (not (GT A B)))

(defun GE (A B)
 (not (LT A B)))

;----------------MANIPULATE POINTS---------------

(defun average-of-points (list-of-points)
 (map 'list '(lambda (a b) (/ a b)) (mean-point-in-facet-2 list-of-points)
                          (make-list 3 :initial-element
                                                  (length list-of-points))))

(defun find-point (X Y Z List-of-points)  ; find all points in list which match
 (let ((result List-of-points)          ; one or more of specified values. values
       (values (list X Y Z)))           ; of 'nil will be ignored. returns a list.
  (loop for Pass in (List 0 1 2)
   do (cond ((not (equal 'nil (nth Pass values)))
```

```
                    (let ((intermediate-result 'nil))
                     (loop for P in result
                       do (cond ((equal-error (nth Pass values)
                                              (nth Pass (send (eval P) :list-format)))
                                 (setf intermediate-result
                                       (adjoin P intermediate-result)))))
                     (setf result intermediate-result)))))
  result))

(defun average-points (Pt1 Pt2) ; find the point 1/2 way between two points
 (map 'list '/ (map 'list '+ (send (eval Pt1) :list-format)
                             (send (eval Pt2) :list-format))
         (make-list 3 :initial-element '2)))



;----------MAKE OR MANIPULATE VECTORS----------

(defun solve-for-t (Plane Line Denom)
 (/ (- (fourth Plane) (apply '+ (map 'list '* Plane
                                     (send (eval(Line-segment-position-vector
                                                 (eval Line))) :list-format)))) Denom))

(defun vector-add-with-t (DV PV Ti)    ; add a direction vector (*T) to a position vector
 (map 'list '+ (send (eval PV) :list-format)
         (map 'list #'(lambda (A) (* A Ti)) (send (eval DV) :list-format))))


;----------MAKE OR MANIPULATE LINES----------


(defun make-line (Point1 Point2)
 (init-line (init-vector '*origin* Point1)
         (init-vector Point1 Point2)))

(defun line-cross-product (L1 L2)  ; take the cross product of direction vectors
 (cross-product (send (eval (line-segment-direction-vector (eval L1))) :list-format)
                (send (eval (line-segment-direction-vector (eval L2))) :list-format))



(defun find-common-point (L1 L2)          ; returns the value of a common point,
  (loop for m in (send (eval L1) :endpoints)  ; if one exists.
        do (loop for n in (send (eval L2) :endpoints)
             when (equal m n)
               do (return-from find-common-point m)))
  'nil)

(defun angle-between-lines (L1 L2)    ; find the smallest angle between two lines
                          ; return NIL for unusual problems
 (let* ((normal-vector (line-cross-product L1 L2))
```

```
        (normal-vector-length (sqrt (abs (+ (* (first normal-vector)
                                               (first normal-vector))
                                            (* (second normal-vector)
                                               (second normal-vector))
                                            (* (third normal-vector)
                                               (third normal-vector)))))))
  (cond ((equal-zero-p normal-vector-length)
         (return-from angle-between-lines 'nil))
        ((or (equal-zero-p (send (eval L1) :length))
         (equal-zero-p (send (eval L2) :length)))
        (return-from angle-between-lines 'nil)))
  (- *PI* (asin (/ normal-vector-length (* (send (eval L1) :length)
                                           (send (eval L2) :length)))))))

(defun find-mid-point (Line)
 (send (eval Line) :midpoint))



;----------MAKE OR MANIPULATE PLANES----------

(defun make-a-plane (point line)   ; define a plane given a point and a line
 (let* ((Obs-line (init-line (init-vector '*origin* point)
                             (init-vector point
                                          (first (send (eval line) :endpoints)))))
        (plane (make-a-normalized-plane Obs-line line)))
  (init-plane plane)))


(defun make-a-normalized-plane (L1 L2)          ; make a plane equation with
                               ; Ao = -1,0,1; first coef is positive
 (let ((un-normalized (line-cross-product L1 L2))  ; normal vector to plane
       (common-point (find-common-point L1 L2))
                                  ; a point in the plane
       (Ao 'nil)                  ; constant in plane equation
       (normalized 'nil))         ; in standard form
  (setf un-normalized (map 'list 'rationalize un-normalized))
  (cond ((null common-point)
         (setf common-point (send (eval (send (eval L1) :start-point)) :list-format)))
        (t (setf common-point (send (eval common-point) :list-format))))
  (setf Ao (apply '+ (mapcar '* common-point un-normalized)))
  (cond ((equal-zero-p Ao)
         (setf normalized
               (map 'list '/ un-normalized (make-list 3 :initial-element
                                                      (first-non-zero un-normalized))))
         (setf normalized (reverse (append (list '0) (reverse normalized)))))

        (t (setf normalized
           (map 'list '/ un-normalized (make-list 3 :initial-element Ao)))
           (setf normalized (reverse (append (list '1) (reverse normalized))))))
  (cond ((GT '0.0 (first-non-zero normalized))
```

```
            (map 'list '* (make-list 4 :initial-element (- 1)) normalized))
           (t 't))
    (setf normalized (map 'list 'rationalize normalized))
    normalized))   ; return the coeffs for the plane


(defun make-vertical-plane (Line)
  (let* ((line-endpoints (send (eval Line) :endpoints))
         (Pt1 (map 'list '+ '(0 0 10)
                          (send (eval (first line-endpoints)) :list-format)))
        (L1 (make-line (init-point Pt1) (second line-endpoints)))
        (L2 (make-line (init-point Pt1) (first line-endpoints))))
    (init-plane (make-a-normalized-plane L1 L2))))


(defun make-z-plane (point)
  (init-plane (make-a-normalized-plane
                          (make-line (init-point
                                  (map 'list '+ (send (eval point ) :list-format)
                                          '(10 0 0)))
                              point)
                          (make-line (init-point
                                  (map 'list '+ (send (eval point) : list-format)
                                          '(0 10 0)))
                              point))))
(defun make-y-plane (point)
  (init-plane (make-a-normalized-plane
                          (make-line (init-point
                                  (map 'list '+ (send (eval point ) :list-format)
                                          '(0 0 10)))
                              point)
                          (make-line (init-point
                                  (map 'list '+ (send (eval point) : list-format)
                                          '(0 10 0)))
                              point))))



(defun make-x-plane (point)
  (init-plane (make-a-normalized-plane
                          (make-line (init-point
                                  (map 'list '+ (send (eval point ) :list-format)
                                          '(10 0 0)))
                              point)
                          (make-line (init-point
                                  (map 'list '+ (send (eval point) : list-format)
                                          '(0 0 10)))
                              point))))


(defun subs-point-into-equation (Plane Point)
```

```
 (apply '+ (map 'list '* (send (eval Point) :list-format) Plane)))


(defun subs-line-into-plane-equation (Line Plane)       ; TRUE if lines lie in plane
 (let* ((endpoints (send (eval Line) :endpoints))
       (point-Aos (list (send (eval plane)
                              :subs-point-into-plane (first endpoints))
                        (send (eval plane)
                              :subs-point-into-plane (second endpoints)))))
  (apply 'and
         (map 'list #'equal-error
              point-Aos
              (make-list 2 :initial-element
                           (fourth (send (eval plane) :list-coeff)))))))

;----------MANIPULATE FACETS----------


(defun find-common-facet (V1 V2)    ; find the first facet that two volumes have in
                         ; common.  Use the assumption that common facets
                         ; will have same name first, else they will have
                         ; the same plane equation.
 (let ((common-facet (first (intersection (volume-facets (eval V1))
                                          (volume-facets (eval V2))))))
  (cond ((not (null common-facet))
         (return-from find-common-facet common-facet))
        ((not (null (facet-connects (eval (first (volume-facets (eval V1)))))))
         (loop for F1 in (volume-facets (eval V1))
           do (cond ((member-p V2 (second (facet-connects (eval F1))))
                     (return-from find-common-facet F1)))))
        (t (loop for F1 in (volume-facets (eval V1))
             do (loop for F2 in (volume-facets (eval V2))
                  do (cond ((send (eval F1) :test-equal F2)
                            (return-from find-common-facet F2)))))))
 'nil)


(defun mean-point-in-facet (Facet)
 (map 'list '(lambda (a b) (/ a b)) (mean-point-in-facet-2 (send (eval Facet) :points))
                        (make-list 3 :initial-element
                                                (length (send (eval Facet) :points)))))

(defun mean-point-in-facet-2 (Points)
 (cond ((null Points) '(0 0 0))
       (t (map 'list '+ (send (eval (first Points)) :list-format)
                        (mean-point-in-facet-2 (rest Points))))))


(defun info-on-facets (list-of-facets)   ; find all points and lines in a list of facets
 (let ((lines 'nil)
       (points 'nil))
```

```
    (loop for F in list-of-facets
      do (let ()
           (setf lines (append (facet-edges (eval F)) lines))
           (setf points (append (send (eval F) :points) points))))
    (setf lines (remove-duplicates lines))
    (setf lines (remove 'nil lines))
    (setf points (remove-duplicates points))
    (setf points (remove 'nil points))
    (list points lines)))


(defun inside-facet-p (point facet)          ; return T iff point is inside
  (let ((horizontal-plane (make-z-plane point))  ; a convex facet
        (vertical-y-plane (make-y-plane point))
        (vertical-x-plane (make-x-plane point))
        (vertical-Ao-x 'nil)
        (vertical-Ao-y 'nil)
        (left-points 'nil)
        (right-points 'nil)
        (edge-points 'nil))

                    ; intercept all edges with horizontal plane,
                    ; plane interception points in left or right
                    ; half, based upon relationship with vertical plane

    (setf vertical-Ao-x (fourth (send (eval vertical-x-plane) :list-coeff)))
    (setf vertical-Ao-y (fourth (send (eval vertical-y-plane) :list-coeff)))
    (loop for L in (facet-edges (eval Facet))
      do (let ((I (find-intercept-point horizontal-plane L))
               (I-Ao-x 'nil)
               (I-Ao-y 'nil))
           (cond ((not (equal 'nil I))
                  (setf I-Ao-y (send (eval vertical-y-plane) :subs-point-into-plane I))
                  (setf I-Ao-x (send (eval vertical-x-plane) :subs-point-into-plane I))
                  (cond ((LT vertical-Ao-x I-Ao-x)
                         (setf right-points (adjoin I right-points)))
                        ((GT vertical-Ao-x I-Ao-x)
                         (setf left-points (adjoin I left-points)))
                        (t (setf edge-points (adjoin I edge-points))))
                  (cond ((LT vertical-Ao-y I-Ao-y)
                         (setf right-points (adjoin I right-points)))
                        ((GT vertical-Ao-y I-Ao-y)
                         (setf left-points (adjoin I left-points)))
                        (t (setf edge-points (adjoin I edge-points))))))))

                    ; test for inclusion by nr of intercept points

    (cond ((or (not (evenp (length left-points)))    ; if either one odd, then point
               (not (evenp (length right-points))))  ; is in facet
           (return-from inside-facet-p 't))
          (t (return-from inside-facet-p 'nil)))))
```

```
;-----------MAKE OR MANIPULATE VOLUMES----------


(defun intersect-all-planes-with-volumes (list-of-planes List-of-volumes)
                    ; intersectal all planes given with all volumes given,
                    ; including resultant volumes from earlier intersections.
                    ; requires input of volumes as: ((volume) (volume) ...)
                    ; resultant volume list is the same format.
 (let ((old-list-of-error-planes 'nil)
       (result-volumes
        (intersect-all-planes-with-volumes-2 List-of-planes List-of-volumes)))
  (loop repeat '1
   do (let ()
          (terpri) (terpri)
          (princ " Re-doing error intercepts: ")
          (prin1 *list-of-error-planes*) (terpri)
          (setf old-list-of-error-planes *list-of-error-planes*)
          (setf *list-of-error-planes* 'nil)
          (setf result-volumes (intersect-all-planes-with-volumes-2
                                    old-list-of-error-planes
                                    result-volumes))))
  result-volumes))


(defun intersect-all-planes-with-volumes-2 (List-of-planes List-of-volumes)
                              ; do all the work for intersect-all-planes-with-volumes
 (let ((templist '()))
  (cond ((null list-of-planes) list-of-volumes)
        (t (loop for V in List-of-volumes
             do (let ((temp 'nil))
                  (setf temp (intersect (car V)
                                        (send (eval (car list-of-planes))
                                              :list-coeff)))
                  (cond ((equal '1 (length temp))
                         (push temp templist))
                        (t (push (list (first temp)) templist)
                           (push (list (second temp)) templist)))))
           (intersect-all-planes-with-volumes-2 (cdr list-of-planes) templist)))))


;-----------PROPERTY LIST MANIPULATIONS----------

(defun reset-point-property-lists (Volume)
 (loop for P in (volume-points (eval Volume))
  do (setf (get P 'lines) 'nil)
  do (setf (get P 'planes) 'nil)
  do (setf (get P 'distance) 'nil)))
```

```
;----------MANIPULATE GLOBAL COUNTERS----------

(defun speed-demon ()
 (terpri) (terpri)                    ; delete *list-of-?????* lists to
 (princ "********SPEED-DEMON-INVOKED********")   ; speed processing. best if
 (terpri) (terpri)                    ; used with static universe methods
 (setf *list-of-vectors* 'nil)           ; if contents of old lists still needed
 (setf *list-of-lines* 'nil)
 (setf *list-of-points* 'nil)
 (setf *list-of-planes* 'nil)
 (make-null-vector)
 (make-origin))


;----------------MORE SPECIFIC STUFF--------------

(defun minimum-distance (lines start-point)
 (let ((best-line (first lines)))
  (cond ((< '1 (length lines))
         (loop for L in (rest lines)
          do (cond ((> (get (send (eval L) :other-end start-point)
                            'distance)
                       (get (send (eval best-line) :other-end start-point)
                            'distance))
                   (setf best-line L))))))
  best-line))



;-----------------------------------------------------
; FIND THE VOLUME(S) CONTAINING A GIVEN POINT
;-----------------------------------------------------

(defun locate-point (point)

          ; return the one, two, or more volumes which contain the point.
          ; multiple volumes are possible if point is on facet or vertex
          ; of a volume

 (let ((list-of-possible-volumes (universe-volumes *universe*))
       (reject-volumes 'nil)
       (x-plane (make-x-plane point))
       (y-plane (make-y-plane point))
       (z-plane (make-z-plane point)))

          ; loop through planes which define point,
          ; removing volumes which do not intersect the planes.
          ; point is located in volume(s) which are left

  (loop for PI in (list x-plane y-plane z-plane)
```

```
    do (let ()

                ; loop through (modified) list of candiate volumes

          (loop for V in list-of-possible-volumes
           do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                          (first (volume-points (eval V)))))

                    (Ao (fourth (send (eval Pl) :list-coeff))))

                        ; see if volume strattles plane

                    (cond ((not (equal-error first-point-Ao Ao))
                           (cond ((point-check-p Pl first-point-Ao Ao V)
                                  (setf reject-volumes
                                        (adjoin V reject-volumes))))))))

                ; remove rejected volumes from possible location of points

          (loop for V in reject-volumes
           do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
          (setf reject-volumes 'nil)))

              ; select actual location of point from final list
              ; of volumes

  (loop for V in list-of-possible-volumes    ; not so good
   do (let ((lines 'nil))
          (loop for F in (volume-facets (eval V))
           do (setf (get F 'center) (init-point (mean-point-in-facet F)))
           do (setf lines (adjoin (make-line Point (get F 'center)) lines)))
          (cond ((lines-strattle-facets-p Lines V)
                 (setf list-of-possible-volumes (remove V list-of-possible-volumes))))))

  list-of-possible-volumes))

(defun locate-point-air (point)

          ; return the one, two, or more air volumes which contain the point.
          ; multiple volumes are possible if point is on facet or vertex
          ; of a volume. Same as locate-point function, except that ground
          ; volumes are removed

 (let ((list-of-possible-volumes (universe-volumes *universe*))
       (reject-volumes 'nil)
       (x-plane (make-x-plane point))
       (y-plane (make-y-plane point))
       (z-plane (make-z-plane point)))

          ; loop through planes which define point,
          ; removing volumes which do not intersect the planes.
```

```
                ; point is located in volume(s) which are left

        (loop for Pl in (list x-plane y-plane z-plane)
         do (let ()

                        ; loop through (modified) list of candiate volumes

                (loop for V in list-of-possible-volumes
                 do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                                (first (volume-points (eval V)))))

                          (Ao (fourth (send (eval Pl) :list-coeff))))

                                ; see if volume strattles plane

                          (cond ((not (equal-error first-point-Ao Ao))
                                 (cond ((point-check-p Pl first-point-Ao Ao V)
                                        (setf reject-volumes
                                              (adjoin V reject-volumes))))))))

                          ; remove rejected volumes from possible location of points

                (loop for V in reject-volumes
                 do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
                (setf reject-volumes 'nil)))

                    ; select actual location of point from final list
                    ; of volumes

        (loop for V in list-of-possible-volumes    ; not so good
         do (let ((lines 'nil))
                (loop for F in (volume-facets (eval V))
                 do (setf (get F 'center) (init-point (mean-point-in-facet F)))
                 do (setf lines (adjoin (make-line Point (get F 'center)) lines)))
                (cond ((lines-strattle-facets-p Lines V)
                       (setf list-of-possible-volumes (remove V list-of-possible-volumes))))))


                    ; remove ground volumes from list

        (loop for V in list-of-possible-volumes
         do (cond ((equal 'ground (volume-composition (eval V)))
                   (setf list-of-possible-volumes (remove V list-of-possible-volumes)))))

        list-of-possible-volumes))



(defun point-in-volume-p (point volume) ; return T iff the point is inside the volume
                                        ; return NIL otherwise
                                        ; code is modified version of locate-point-air
```

```
(let ((list-of-possible-volumes (list volume))
      (reject-volumes 'nil)
      (x-plane (make-x-plane point))
      (y-plane (make-y-plane point))
      (z-plane (make-z-plane point)))

            ; see if point is a vertex, or in a facet of the volume

  (cond ((member-p point (volume-points (eval volume)))
         (return-from point-in-volume-p 't)))
  (loop for F in (volume-facets (eval volume))
   do (cond ((inside-facet-p point F)
             (return-from point-in-volume-p 't))))

            ; loop through planes which define point,
            ; removing volumes which do not intersect the planes.
            ; point is located in volume(s) which are left

  (loop for Pl in (list x-plane y-plane z-plane)
   do (let ()

                ; loop through (modified) list of candiate volumes

          (loop for V in list-of-possible-volumes
           do (let ((first-point-Ao (send (eval Pl) :subs-point-into-plane
                                          (first (volume-points (eval V)))))

                    (Ao (fourth (send (eval Pl) :list-coeff))))

                        ; see if volume strattles plane

                  (cond ((not (equal-error first-point-Ao Ao))
                         (cond ((point-check-p Pl first-point-Ao Ao V)
                                (setf reject-volumes
                                      (adjoin V reject-volumes))))))))

                  ; remove rejected volumes from possible location of points

          (loop for V in reject-volumes
           do (setf list-of-possible-volumes (remove V list-of-possible-volumes)))
          (setf reject-volumes 'nil)))

  (cond ((null list-of-possible-volumes)              ; exit condition
         (return-from point-in-volume-p 'nil)))

  't))


(defun point-check-p (Plane Basis-point-Ao Ao Volume)
 (loop for P in (rest (volume-points (eval Volume)))
```

```
  do (let ((next-point-Ao (send (eval Plane) :subs-point-into-plane P)))
          (cond ((equal next-point-Ao Ao)
                    (return-from point-check-p 'nil))
                ((or (and (GT Ao Next-point-Ao)
                          (LT Ao basis-point-Ao))
                     (and (LT Ao Next-point-Ao)
                          (GT Ao basis-point-Ao)))
                 (return-from point-check-p 'nil)))))
  't)



(defun lines-strattle-facets-p (Lines Volume)
 (loop for L in Lines
  do (loop for F in (volume-facets (eval Volume))
          do (cond ((send (eval L) :strattle-plane-p F)
                    (return-from lines-strattle-facets-p 't)))))
 'nil)



;-------------------------------------------------------
; PRINT GOOD-TO-KNOW INFO CONCERNING THE STATE
; OF THE *UNIVERSE* INTO A DISK FILE
;-------------------------------------------------------

(defun dump-volumes (list-of-volumes)
 (setq *output-stream* (open "exp3:lewis;run2" :direction :output))
 (print "sending data to file (run2)...")
 (loop for V in List-of-volumes
   do (let ()
           (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
           (print-volumes (list V))
           (terpri *output-stream*)
           (print-points (volume-points (eval V)))
           (terpri *output-stream*)
           (print-lines (volume-edges (eval V)))
           (terpri *output-stream*)
           (print-facets (volume-facets (eval V)))))
 (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
 (close *output-stream*)
 (print "Done.") 'nil)


(defun dump-path (path-name)
 (setq *output-stream* (open "exp3:lewis;path-dump" :direction :output))
 (print "sending path data to file (path-dump)...")
 (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
 (print-path path-name)
 (terpri *output-stream*)
 (print-points (path-points (eval path-name)))
```

```
 (terpri *output-stream*)
 (print-lines (path-lines (eval path-name)))
 (terpri *output-stream*)
 (print-facets (path-facets (eval path-name)))
 (terpri *output-stream*) (terpri *output-stream*) (terpri *output-stream*)
 (close *output-stream*)
 (print "Done.")
'nil)


;*****************************************************************************
;
;;;
;;; PRINT FLAVOR FUNCTIONS                                    20 May 1988
;;;
;*****************************************************************************
;



(defun print-points (List)
 (cond ((null List))
       (t (terpri *output-stream*)
          (prin1 "name: " *output-stream*)
          (prin1 (car List) *output-stream*)
          (send (eval (car List)) :print)
          (print-points (cdr List)))))

(defun print-vectors (List)
 (cond ((null List))
       (t (terpri *output-stream*)
          (prin1 "name: " *output-stream*)
          (prin1 (car List) *output-stream*)
          (send (eval (car List)) :print)
          (print-vectors (cdr List)))))

(defun print-lines (List)
 (cond ((null List))
       (t (terpri *output-stream*)
          (prin1 "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
          (send (eval (car List)) :print)
          (print-lines (cdr List)))))

(defun print-facets (List)
 (cond ((null List))
       (t (terpri *output-stream*)
          (prin1 "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
          (send (eval (car List)) :print)
          (print-facets (cdr List)))))
```

```
(defun print-volumes (List)
 (cond ((null List))
       (t (terpri *output-stream*)
          (prin1  "name:" *output-stream*)
          (prin1 (car List) *output-stream*)
          (send (eval (car List)) :print)
          (print-volumes (cdr List)))))

(defun print-path (name)
 (terpri *output-stream*)
 (princ "name: " *output-stream*) (prin1 name *output-stream*)
 (princ "start-point: " *output-stream*)
 (prin1 (path-start-point (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "end-point: "*output-stream*)
 (prin1 (path-end-point (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "volumes: "*output-stream*)
 (prin1 (path-volumes (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "facets: "*output-stream*)
 (prin1 (path-facets (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "lines: "*output-stream*)
 (prin1 (path-lines (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "points: "*output-stream*)
 (prin1 (path-points (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "length: "*output-stream*)
 (prin1 (path-length (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "total K values: "*output-stream*)
 (prin1 (path-total-K (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "maximum detection probability: "*output-stream*)
 (prin1 (path-max-detection-probability (eval name)) *output-stream*)
 (terpri *output-stream*)
 (princ "average detection probability: "*output-stream*)
 (prin1 (path-ave-detection-probability (eval name)) *output-stream*)
 (terpri *output-stream*))
```

FILE NAME: Interception.lisp

```
;;; -*- Mode:Lisp; Syntax: Common-lisp -*-
;*****************************************************************************
;
;  FUNCTIONS TO INTERCEPT A VOLUME WITH A PLANE        D.H.LEWIS    27May88
;
;  ------------------------------------------------------------------
;
; These functions intercept planes with volumes and lines with planes.  Multiple
; tests are performed to ensure proper construction of new volumes.  Facets are
; rebuilt each time.
;
; Main functions:  INTERSECT (VOLUME PLANE)
;                  FIND-INTERCEPT-POINT (PLANE LINE)
;
; Other functions: GET-INTERCEPT-POINT (PLANE LINE T-INTERCEPT)
;                  PUT-LINE-IN-CORRECT-HALF (LINE PLANE)
;                  PUSH-ENDPOINTS (LINE VOLUME)
;                  MAKE-NEW-DIVIDING-LINES (VOLUME OLDPOINTS NEW-POINTS)
;                  NEW-VALID-LINE (POINT1 POINT2 VOLUME)
;                  IN-FACET-P (POINT1 POINT2 FACET)
;                  OUTSIDE-VOLUME (LINE VOLUME)
;                  DENOM-IN-INTERCEPT (PLANE LINE)
;                  GET-INTERCEPT-POINT-2 (LINE T-INTERCEPT)
;
;*****************************************************************************

(defvar *lines-in-intercept-plane* 'nil)
(defvar *large-integer* '1e4)
(defvar *list-of-error-planes* 'nil)    ; used to correct errors in interceptions

(defun intersect (Volume Plane)
  (let ((old-precision *precision*)
        (bad-euler-flag 't)
        (new-volume1 'nil)
        (new-volume2 'nil)
        (facet-planes 'nil)
        (intercept-plane 'nil))
    (terpri) (princ "intersecting: ") (prin1 (list Volume Plane))
    (princ " --- Result: ")
    (setf *lines-in-intercept-plane* 'nil)
    (cond ((bad-intersect-preconditions-p Volume Plane); check for degenerate conditions
           (return-from intersect (list volume))))
    (setf intercept-plane (init-plane Plane))
    (loop for F in (volume-facets (eval Volume))        ; get all planes used
      do (setf facet-planes (adjoin (init-plane (send (eval F) :list-coeff))
                                    facet-planes)))
    (setf facet-planes (adjoin intercept-plane facet-planes))
    (setf facet-planes (remove-duplicates facet-planes))
    (loop until (or (not bad-euler-flag) (> *precision* (* '5 old-precision)))
```

```
do (let ()
                    ; clear standard volumes before use (or reuse)
                    ; and set common values

    (send *above* :clear)
    (setf (volume-visibility *above*) (volume-visibility (eval Volume)))
    (setf (volume-composition *above*) (volume-composition (eval Volume)))
    (send *below* :clear)
    (setf (volume-visibility *below*) (volume-visibility (eval Volume)))
    (setf (volume-composition *below*) (volume-composition (eval Volume)))

              ; conduct intercept

    (let ((List-of-new-points 'nil)
          (list-of-old-points 'nil))
     (loop for P in (volume-points (eval Volume))
      do (setf (get P 'lines) 'nil))
                          ;intersect each line of volume
     (loop for Line in (Volume-Edges (eval Volume))
      do (let ((new-point (find-intercept-point intercept-plane Line)))
           (cond ((equal new-point 'nil)
                  (cond ((not (subs-line-into-plane-equation Line
                                                    intercept-plane))
                         (put-line-in-correct-half
                              Line
                              (first (send (eval Line) :endpoints))
                              intercept-plane))))
                 ((member-p new-point (Volume-points (eval Volume)))
                  (pushnew new-point list-of-old-points)
                  (put-line-in-correct-half
                         Line new-point intercept-plane))
                 (t (pushnew new-point List-of-new-points)
                  (place-intercept-point Plane Line New-point)))))

    (make-new-dividing-lines Volume List-of-new-points list-of-old-points))
       (cond ((not (simple-volume-test-p))       ; check degenerate cases
              (setf *precision* old-precision)
              (return-from intersect (list volume))))

                    ; build new facets in best way possible

       (cond ((not *not-convex-volumes*)         ; do convex facets
              (make-facets facet-planes *above*)  ; quick facet builder
              (make-facets facet-planes *below*)
              (cond ((not (check-eulers-relation-p))
                     (setf (volume-facets *above*) 'nil)
                     (setf (volume-facets *below*) 'nil)
                     (make-all-facets *above*)    ; slow facet builder
                     (make-all-facets *below*))))
             (t (make-all-facets *above*)        ; do non-convex facets
              (make-all-facets *below*)))
```

```
            (cond ((null (intersection (volume-facets *above*)
                                       (volume-facets *below*)))
                   (force-facet plane)))

            (cond ((not (check-eulers-relation-p))
                   (setf *precision* (* *precision* '2.0)))
                  (t (setf bad-euler-flag 'nil)))))

   (cond ((not bad-euler-flag)
          (setf new-volume1 (make-volume-name))    ; give legitimate names to new
          (setf new-volume2 (make-volume-name))    ; volumes
          (send *above* :make-equal new-volume1)
          (send *below* :make-equal new-volume2)
          (push  new-volume1 *list-of-volumes*)
          (push  new-volume2 *list-of-volumes*)
          (setf *precision* old-precision)
          (prin1 (intersection (volume-facets (eval new-volume1))
                               (volume-facets (eval new-volume2))))
          (return-from intersect (list new-volume1 new-volume2)))  ; return new volumes
         (t (setf *precision* old-precision)
          (setf *list-of-error-planes*
                (adjoin intercept-plane *list-of-error-planes*))
          (return-from intersect (list Volume))))))

(defun subs-point-into-plane (Pt Plane)
 (send (eval Plane) :subs-point-into-plane Pt))

(defun bad-intersect-preconditions-p (Volume Plane)  ; test for null plane (0 0 0 0)
                                     ; and facet intercept if convex
  (cond ((equal *zero-vector* (map 'list '* plane *one-vector*))
         (princ "nil (early 1)")
         (return-from bad-intersect-preconditions-p 't))
;        ((not *not-convex-volumes*)
;         (loop for F in (volume-facets (eval Volume))
;           do (cond ((send (eval F) :test-equal (init-plane Plane))
;                     (princ "nil (early 2)")
;                     (return-from bad-intersect-preconditions-p 't)))))
         )
 'nil)

(defun find-intercept-point (plane line) ; find intercept point of plane and line
                         ; segment, if it exists.  return NIL
                         ; if not exist
 (let ((denom (rationalize (denom-in-intercept plane line)))
       (t-intercept 'nil)
       (l-point 'nil))
  (cond ((not (equal-zero-p denom))
         (setf t-intercept (rationalize (solve-for-t
                                         (send (eval plane) :list-coeff)
                                         line
```

```
                                                    denom)))
            (setf I-point (get-intercept-point-2 line t-intercept))))
   I-point))

(defun denom-in-intercept (plane line) ; find the denominator in intercept equation
  (apply '+ (map 'list '* (send (eval plane) :list-coeff)
                          (map 'list 'rationalize
                               (send (eval (line-segment-direction-vector
                                             (eval line))) :list-format)))))

(defun get-intercept-point-2 (line t-intercept)
                          ; return the name of a valid intercept point
 (let ((I 'nil)
       (I-list 'nil))
  (cond ((not (or (GT t-intercept (line-segment-t-max (eval line)))
                  (LT t-intercept '0.0)))
         (setf I-list (vector-add-with-t
                                (line-segment-direction-vector (eval line))
                                (line-segment-position-vector (eval line))
                                t-intercept))
         (setf I (init-point I-list))))
  I))

(defun place-intercept-point (Plane Line I) ; divide old line at I, build new lines
 (let ((L1 'nil)                    ; and put each in right resultant volume
       (L2 'nil))
         (setf (get I 'lines) Line)
         (pushnew I (volume-points *above*))
         (pushnew I (volume-points *below*))
         (setf L1 (make-line I (first (send (eval Line) :endpoints))))
         (setf L2 (make-line I (second (send (eval Line) :endpoints))))
         (setf (line-segment-characteristics (eval L1))     ; ridge is still a ridge
                          (line-segment-characteristics (eval Line)))
         (setf (line-segment-characteristics (eval L2))
                          (line-segment-characteristics (eval Line)))
         (cond ((LT (fourth Plane)    ; which volume to put new line L1?
                    (subs-point-into-equation Plane (car (send (eval Line) :endpoints
                                                                        ))))
                (pushnew L1 (volume-edges *above*))
                (push-endpoints L1 '*above*))
               ((GT (fourth Plane)
                    (subs-point-into-equation Plane (car (send (eval Line) :endpoints
                                                                        ))))
                (pushnew L1 (volume-edges *below*))
                (push-endpoints L1 '*below*))
               (t ))
         (cond ((LT (fourth Plane)    ; Which volume to put new line L2?
                    (subs-point-into-equation Plane (cadr (send (eval Line) :endpoints
                                                                        ))))
                (pushnew L2 (volume-edges *above*))
                (push-endpoints L2 '*above*))
```

```
                ((GT (fourth Plane)
                   (subs-point-into-equation Plane (cadr (send (eval Line) :endpoints
                                                                          ))))
                 (pushnew L2 (volume-edges 'below'))
                 (push-endpoints L2 ''below')))))

(defun put-line-in-correct-half (Line Point Plane) ; put a preexisting volume line
                                ; into the correct resultant volume
  (let ((Plane-Ao (fourth (send (eval Plane) :list-coeff)))
        (other-point (send (eval Line) :other-end Point)))
    (cond ((GT (send (eval Plane) :subs-point-into-plane other-point)
               Plane-Ao)
           (pushnew Line (volume-edges 'above'))
           (push-endpoints Line ''above'))
          (t (pushnew Line (volume-edges 'below'))
             (push-endpoints Line ''below')))))


(defun push-endpoints (Line Volume)
  (pushnew (first (send (eval Line) :endpoints)) (volume-points (eval Volume)))
  (pushnew (second (send (eval Line) :endpoints)) (volume-points (eval Volume))))


(defun make-new-dividing-lines (Volume List-new-points List-old-points)
   (loop for P1 in List-new-points    ; handle case where no pre-exitant points involved
         do (loop for P2 in List-new-points
              do (cond ((not (equal P1 P2))
                        (new-valid-line P1 P2 Volume)))))

   (loop for P1 in List-old-points    ; add pre-exitant lines and points
     do (loop for P2 in List-old-points  ;to new volumes
           do (cond ((not (equal P1 P2))
                     (new-valid-line P1 P2 Volume) ;make new connecting lines
                                    ;then find old ones
                     (loop for Line in (volume-edges (eval Volume))
                           do (let ((endpoint1 (first (send (eval Line) :endpoints)))
                                    (endpoint2 (second (send (eval Line) :endpoints))))
                                (cond ((and (or (equal P1 endpoint1)
                                                (equal P1 endpoint2))
                                            (or (equal P2 endpoint1)
                                                (equal P2 endpoint2)))
                                       (push-endpoints Line 'above')
                                       (push-endpoints Line 'below')
                                       (pushnew Line (volume-edges 'above'))
                                       (pushnew Line (volume-edges 'below'))
                                       (pushnew Line 'lines-in-intercept-plane')))))))))

   (loop for P-new in List-new-points      ; add new lines connecting old and new
         do (loop for P-old in List-old-points  ; points to new volumes
              do (new-valid-line P-new P-old Volume))))
```

```
(defun new-valid-line (P1 P2 Volume) ; make a new (and valid) line between P1 and P2
                      ; which lies inside (or along an edge) of Volume
 (loop for F1 in (volume-facets (eval Volume))
   do (cond ((in-facet-p P1 P2 F1)
             (let ((Line (make-line P1 P2)))
                  (cond ((not (outside-volume Line Volume))
                         (push-endpoints Line *above*)
                         (push-endpoints Line *below*)
                         (pushnew Line (volume-edges *above*))
                         (pushnew Line (volume-edges *below*))
                         (pushnew Line *lines-in-intercept-plane*)))))))


(defun simple-volume-test-p ()
 (cond ((or (or (> '3 (length (volume-points *above*)))
                (> '3 (length (volume-points *below*))))
            (or (> '5 (length (volume-edges *above*)))
                (> '5 (length (volume-edges *below*)))))
        (princ "nil (late 1)")
        (return-from simple-volume-test-p 'nil)))
  't)

(defun check-eulers-relation-p ()
 (cond ((or (not (equal '2 (+ (length (volume-points *above*))
                              (length (volume-facets *above*))
                              (- '0 (length (volume-edges *above*))))))
            (not (equal '2 (+ (length (volume-points *below*))
                              (length (volume-facets *below*))
                              (- '0 (length (volume-edges *below*)))))))
        (princ " Violated Eulers relation ") (prin1 *precision*)
        (terpri)
        (princ "                                    ")
        (return-from check-eulers-relation-p 'nil)))
 't)

(defun make-facets (planes volume)
 (loop for P in planes              ; clear plane properties
   do (setf (get P 'edges) 'nil))

 (loop for P in planes              ; find which lines lie in which planes
   do (loop for E in (volume-edges (eval Volume))
         do (cond ((subs-line-into-plane-equation E P)
                   (setf (get P 'edges) (adjoin E (get P 'edges)))))))

 (loop for P in planes              ; build legitimate facets
   do (cond ((and (not (null (get P 'edges)))
                  (<= '3 (length (get P 'edges))))
             (setf (volume-facets (eval Volume))
                   (adjoin (init-facet-2 (list (get P 'edges) P))
                           (volume-facets (eval Volume)))))))
```

```
 (loop for P in planes                ; clear plane properties
  do (setf (get P 'edges) 'nil)))


(defun force-facet (Plane)   ; force a facet to exist, if all else fails
 (let* ((lines-in-facet *lines-in-intercept-plane*)
        (forced-facet (init-facet-2 (list lines-in-facet (init-plane Plane)))))
  (setf (volume-facets *above*) (adjoin forced-facet (volume-facets *above*)))
  (setf (volume-facets *below*) (adjoin forced-facet (volume-facets *below*)))
  (princ " Forced ")))


(defun in-facet-p (P1 P2 F)        ; return T iff points P1 and P2  are inside facet F
 (cond ((and (or (member-p (get P1 'lines) (facet-edges (eval F)))
                 (member-p P1 (send (eval F) :points)))
             (or (member-p (get P2 'lines) (facet-edges (eval F)))
                 (member-p P2 (send (eval F) :points))))
        (return-from in-facet-p 't))
       (t (return-from in-facet-p 'nil))))


(defun outside-volume (Line Volume)  ; return T iff line is outside the volume
                       ; do only if dealing with ground volumes or
                       ; non-convex air volumes
 (cond ((or *not-convex-volumes*
            (equal 'ground (volume-composition (eval volume))))
        (let ((mid-point (init-point (send (eval line) :midpoint))))
         (cond ((point-in-volume-p mid-point volume)
                (return-from outside-volume 'nil))
               (t (return-from outside-volume 't)))))
       (t (return-from outside-volume 'nil))))
```

FILE NAME: Camera.lisp

```
;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;*****************************************************************************
;
;;;
;;; FLAVORS FOR 3-D DISPLAY OF VOLUMES     ;Written by Dr Sehung Kwak
;;;                                        ;for CS4452
;;; THESIS              D.H. Lewis                    18 May 1988
;;;
;*****************************************************************************
;


(defflavor Graphic
           (node-list
            polygon-list
            transformed-node-list
            H-matrix)
           ()
 :inittable-instance-variables
 :settable-instance-variables
 :gettable-instance-variables
 :outside-accessible-instance-variables)

(defmethod (Graphic :translate-and-euler-angle-transform)
        (x y z azimuth elevation roll)
 (let ()
    (setf H-matrix
          (homogeneous-transform azimuth elevation roll x y z))
    (setf transformed-node-list
          (mapcar #'(lambda (node-location) (post-multiply H-matrix node-location))
                                                  node-list))))

(defmethod (graphic :get-node-polygon-list) ()
 (list transformed-node-list polygon-list))

(defmethod (graphic :initialize) ()
 (setf node-list (send self :make-node-list))
 (setf polygon-list (send self :make-polygon-list))
 (setf transformed-node-list node-list)
 (setf H-matrix (unit-matrix 4)))

(defmethod (graphic :get-transformed-node-list) ()
 transformed-node-list)

;*****************************************************************************
;
;;;
;;; CAMERA FLAVOR AND METHODS TO USE GRAPHIC FLAVOR
;;;
;;;                          ;Written by Dr Sehung Kwak
;;;                          ;for CS4452
;;;
```

```
;;; THESIS          D.H. Lewis          18 May 1988
;*****************************************************************************

(defflavor camera
        (H-matrix
         image-distance
         previous-point
         *camerwindow*
         scale)
        ()
 :initable-instance-variables
 :gettable-instance-variables
 :outside-accessible-instance-variables)



(defmethod (camera :initialize)
          ()
 (setf H-matrix (unit-matrix 4))
 (setf image-distance *image-distance*)
 (setf scale *scale*)
 (setf *camerwindow*
        (tv:make-window 'tv:window
                        :blinker-p nil
                        :position *window-upper-left-corner*
                        :inside-width *window-width*
                        :inside-height *window-height*
                        :name "VOLUME WINDOW"
                        :save-bits t
                        :expose-p t)))

(defmethod (camera :initialize-B)    ; for advanced functions
          (window-stats)
 (setf H-matrix (unit-matrix 4))
 (setf image-distance *image-distance*)
 (setf scale *scale*)
 (setf *camerwindow*
        (tv:make-window 'tv:window
                        :blinker-p nil
                        :position (list (first window-stats)
                                        (second window-stats))
                        :inside-width (third window-stats)
                        :inside-height (fourth window-stats)
                        :name (fifth window-stats)
                        :save-bits t
                        :expose-p t)))

(defmethod (camera :move)
          (x y z azimuth elevation roll)
 (setf H-matrix
        (homogeneous-transform azimuth elevation roll x y z)))
```

```
(defmethod (camera :take-picture)
           (solid-object)
 (let* ((node-polygon-list
          (send (eval solid-object) :get-node-polygon-list))
        (node-vector (send self :convert-list-of-lists-to-vector
                           (first node-polygon-list)))
        (polygon-list (second node-polygon-list)))
;  (send *camerwindow* :refresh)      ; don't need for multiple shots
  (dolist (polygon polygon-list)
   (send self :draw-polygon polygon node-vector))))


(defmethod (camera :draw-polygon)
           (polygon node-vector)
 (let ((first-point (first polygon))
       (rest-points (cdr polygon)))
  (send self :move-pen (elt node-vector first-point))
  (dolist (point rest-points)
   (send self :draw-line (elt node-vector point)))
  (send self :draw-line (elt node-vector first-point))))


(defmethod (camera :move-pen)
           (point)
 (setf previous-point (send self :screen-transform point)))


(defmethod (camera :draw-line)
           (next-point)
 (let ((current-point (send self :screen-transform next-point)))
  (send self :draw-line-on-screen previous-point current-point)
  (setf previous-point current-point)))


(defmethod (camera :draw-line-on-screen)
           (from-point to-point)
 (send *camerwindow* :draw-line
             (first from-point) (second from-point)
             (first to-point) (second to-point)
             *thickness*))


(defmethod  (camera :convert-list-of-lists-to-vector)
            (list-of-lists)
 (eval (cons 'vector
             (mapcar '(lambda (component-list)
```

```
                    (cons 'list component-list))
                list-of-lists))))


(defmethod (camera :screen-transform)
          (point)
 (let* ((point-on-camerspace
          (post-multiply
           (matrix-multiply
            H-matrix
            '((1 0 0 0) (0 1 0 0) (0 0 -1 0) (0 0 0 1))
            )
           point))
         (x (first point-on-camerspace))
         (y (second point-on-camerspace))
         (z (third point-on-camerspace)))
  (cond ((equal 0.0 z) (setf z 0.00001))
          (t))
   (list (+ (round (* scale (/ (* image-distance x) z))) (/ *window-width* 2))
          (-  (/ *window-height* 2) (round (* scale (/ (* image-distance y) z)))))))


(defmethod (camera :kill-camera-window)
           ()
 (send *camerwindow* :kill))

(defun take-picture (Camera List-of-objects)
 (send (eval Camera) :initialize)
 (send (eval Camera) :move '2000 '2000 '2000 '0 '0.5 '0.75)
 (loop for V in List-of-objects
         do (send (eval V) :initialize)
         do (send (eval V) :translate-and-euler-angle-transform '-2500
                  '-2000 '-2000 '0.6 '0.6 '-0.1)
         do (send (eval Camera) :take-picture V)))


;-------------------------------------------------------
; advanced camera functions              D.H. Lewis
;-------------------------------------------------------

(defvar *window-width* 700)
(defvar *window-height* 400)
(defvar *window-upper-left-corner* '(10 10))
(defvar *scale* 5)
(defvar *image-distance* 120)
(defvar *thickness* '5)                ; line thickness, in pixels

(defvar *ideal*)
(defvar *low-left-front*)
(defvar *high-left-front*)
(defvar *low-right-front*)
```

```
(defvar *right-side*)
(defvar *left-rear-3/4*)
(defvar *top*)
(defvar *display-stats*)
(defvar *nikon-1*)
(defvar *nikon-2*)
(defvar *nikon-3*)
(defvar *nikon-4*)
(defvar *nikon-5*)
(defvar *nikon-6*)
(defvar *list-of-cameras*)
(defvar *window-stats*)

(defun make-cameras ()
 (setf *nikon-1* (make-instance 'camera))
 (setf *nikon-2* (make-instance 'camera))
 (setf *nikon-3* (make-instance 'camera))
 (setf *nikon-4* (make-instance 'camera))
 (setf *nikon-5* (make-instance 'camera))
 (setf *list-of-cameras* '(*nikon-1* *nikon-2* *nikon-3* *nikon-4* *nikon-5*))
 (setf *ideal* '(100.0 200.0 4000.0 0.0 0.50 1.3 1.0 1.0 1.0 -1.5 0.3 0.0))
 (setf *low-left-front* '(100.0 200.0 4000.0 0.0 0.50 1.0 1.0 1.0 -1.5 0.0 0.0 0.0))
 (setf *high-left-front* '(-100.0 500.0 4000.0 0.0 0.50 1.0 1.0 -1.5 0.0 0.0 0.0 0.0))
 (setf *low-right-front* '(100.0 100.0 4000.0 0.0 0.5 1.5 1.0 1.0 1.0 0.0 0.0 0.0))
 (setf *right-side* '(500.0 0.0 4000.0 0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0))
 (setf *left-rear-3/4* '(-500.0 0.0 4000.0 0.0 0.0 1.0 1.0 1.0 1.0 0.0 0.0 0.0))
 (setf *top* '(0.0 500.0 4000.0 0.0 0.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0))
 'nil)


;----------MAIN FOUR WINDOW DISPLAY----------

(defun display ()
  (setf *window-stats* '('nil
                         (10 20 700 400 "air-volumes" 5 140)
                         (10 440 200 200 "top-view: ground" 5 60)
                         (260 440 200 200 "same-view: ground" 5 60)
                         (510 440 200 200 "full-view: ground" 5 60)))
  (setf *display-stats* (list 'nil
                              *high-left-front*
                              *top*
                              *high-left-front*
                              *ideal*))
  (let ((air-volumes 'nil)
        (ground-volumes 'nil)
        (objects 'nil))
   (loop for V in (universe-volumes *universe*)
    do (cond ((equal 'ground (volume-composition (eval V)))
              (setf ground-volumes (adjoin V ground-volumes)))
             (t (setf air-volumes (adjoin V air-volumes)))))
   (loop for Obs in (universe-observers *universe*)
```

```
    do (setf ground-volumes (adjoin Obs ground-volumes))
    do (setf air-volumes (adjoin Obs air-volumes)))
   (setf objects (list 'nil air-volumes ground-volumes ground-volumes ground-volumes))
   (loop for N in '(1 2 3 4)
    do (take-picture-4 (nth N *list-of-cameras*)
                       (nth N *window-stats*)
                       (nth N objects)
                       (nth N *display-stats*))))

 'nil)


;--------------DISPLAY VISIBLE AIR VOLUMES (3 WINDOWS)----------------------

(defun display-visible (observer)
 (setf *window-stats* '('nil
                        (10 20 700 400 "visible-air-volumes" 5 140)
                        'nil
                        (260 440 200 200 "same-view-ground" 5 60)
                        (510 440 200 200 "full-view-ground" 5 60)))
 (setf *display-stats* (list 'nil
                             *high-left-front*
                             'nil
                             *high-left-front*
                             *ideal* ))
  (let ((visible-volumes 'nil)
        (ground-volumes 'nil)
        (objects 'nil))
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes))
             (setf visible-volumes (adjoin V visible-volumes)))
        ((member-p observer (volume-visibility (eval V)))
             (setf visible-volumes (adjoin V visible-volumes)))))
  (loop for Obs in (universe-observers *universe*)
    do (setf ground-volumes (adjoin Obs ground-volumes))
    do (setf visible-volumes (adjoin Obs visible-volumes)))
  (setf objects (list 'nil visible-volumes 'nil ground-volumes ground-volumes))
  (loop for N in '(1 3 4)
   do (take-picture-4 (nth N *list-of-cameras*)
                      (nth N *window-stats*)
                      (nth N objects)
                      (nth N *display-stats*))))
 'nil)


;---------------DISPLAY NON VISIBLE AIR VOLUMES (3 WINDOWS)-----------------

(defun display-not-visible (observer)
 (setf *window-stats* '('nil
                        (10 20 700 400 "non-visible-air-volumes" 5 140)
```

```
                              'nil
                              (260 440 200 200 "same-view-ground" 5 60)
                              (510 440 200 200 "full-view-ground" 5 60)))
 (setf *display-stats* (list 'nil
                                  *high-left-front*
                                  'nil
                                  *high-left-front*
                                  *ideal* ))
  (let ((invisible-volumes 'nil)
        (ground-volumes 'nil)
        (objects 'nil))
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes))
             (setf invisible-volumes (adjoin V invisible-volumes)))
            ((not (member-p observer (volume-visibility (eval V))))
             (setf invisible-volumes (adjoin V invisible-volumes)))))
  (loop for Obs in (universe-observers *universe*)
     do (setf ground-volumes (adjoin Obs ground-volumes))
     do (setf invisible-volumes (adjoin Obs invisible-volumes)))
  (setf objects (list 'nil invisible-volumes 'nil ground-volumes ground-volumes))
  (loop for N in '(1 3 4)
   do (take-picture-4 (nth N *list-of-cameras*)
                              (nth N *window-stats*)
                              (nth N objects)
                              (nth N *display-stats*))))
 'nil)


;---------DISPLAY SELECTED VOLUMES AND THE GROUND (2 WINDOWS)----------
-

(defun display-volumes (list-of-volumes)
 (setf *window-stats* '('nil
                              (10 20 700 400 "desired-volumes" 5 140)
                              'nil
                              (510 440 200 200 "full-view-ground" 5 60)
                              'nil))
 (setf *display-stats* (list 'nil
                                  *high-left-front*
                                  'nil
                                  *high-left-front*
                                  'nil))
 (let ((objects 'nil)
       (ground-volumes 'nil))
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'ground (volume-composition (eval V)))
                 (setf ground-volumes (adjoin V ground-volumes)))))
  (loop for Obs in (universe-observers *universe*)
     do (setf ground-volumes (adjoin Obs ground-volumes)))
  (setf objects (list 'nil
```

```
                              list-of-volumes
                              'nil
                              ground-volumes
                              'nil))
   (loop for N in '(1 3)
     do (take-picture-4 (nth N *list-of-cameras*)
                              (nth N *window-stats*)
                              (nth N objects)
                              (nth N *display-stats*))))
 'nil)


;----------------DISPLAY PATH AND GROUND (3 WINDOWS)----------------------

(defun display-path (path-name)
 (setf *window-stats* '('nil
                              (10 20 600 380 "Path-over-ground" 5 140)
                              (10 420 600 290 "Alternate-view " 5 140)
                              (618 200 200 200 "Top-view" 5 60)
                              (618 420 200 200 "Low-side view" 5 60)))
 (setf *display-stats* (list 'nil
                                 *ideal*
                                 *high-left-front*
                                 *top*
                                 *right-side*))
 (let ((objects 'nil)
       (ground-volumes 'nil))
  (loop for V in (universe-volumes *universe*)
    do (cond ((equal 'ground (volume-composition (eval V)))
              (setf ground-volumes (adjoin V ground-volumes)))))
  (setf ground-volumes (append (universe-observers *universe*) ground-volumes))
  (setf objects (list 'nil
                              (adjoin path-name ground-volumes)
                              (adjoin path-name ground-volumes)
                              (adjoin path-name ground-volumes)
                              (adjoin path-name ground-volumes)))
  (loop for N in '(1 2 3 4)
    do (take-picture-4 (nth N *list-of-cameras*)
                              (nth N *window-stats*)
                              (nth N objects)
                              (nth N *display-stats*))))
 'nil)


(defun display-paths (list-of-paths)
 (setf *window-stats* '('nil
                              (10 20 600 380 "Paths-over-ground" 5 140)
                              (10 420 600 290 "Alternate-view " 5 140)
                              (618 200 200 200 "Top-view" 5 60)
                              (618 420 200 200 "Low-side view" 5 60)))
 (setf *display-stats* (list 'nil
```

```
                                *ideal*
                                *high-left-front*
                                *top*
                                *right-side*))
 (let ((objects 'nil)
       (ground-volumes 'nil))
  (loop for V in (universe-volumes *universe*)
   do (cond ((equal 'ground (volume-composition (eval V)))
             (setf ground-volumes (adjoin V ground-volumes)))))
  (setf ground-volumes (append (universe-observers *universe*) ground-volumes))
  (setf objects (list 'nil
                      (append list-of-paths ground-volumes)
                      (append list-of-paths ground-volumes)
                      (append list-of-paths ground-volumes)
                      (append list-of-paths ground-volumes)))
  (loop for N in '(1 2 3 4)
   do (take-picture-4 (nth N *list-of-cameras*)
                      (nth N *window-stats*)
                      (nth N objects)
                      (nth N *display-stats*))))
 'nil)


;----------SIMPLE CAMERA ORDERS FOR A PICTURE (MANUAL CONTROL)----------

(defun take-picture-3 (List-of-objects x y z az roll rot ox oy oz oaz oroll orot)
 (let ((Camera '*nikon*))
 (send (eval Camera) :initialize)
 (send (eval Camera) :move (- x) (- y) z az roll rot )
 (loop for V in List-of-objects
       do (send (eval V) :initialize)
       do (send (eval V) :translate-and-euler-angle-transform ox oy oz oaz oroll orot)
       do (send (eval Camera) :take-picture V))))


;-----ADVANCED CAMERA ORDERS FOR A PICTURE (SEMI-AUTOMATIC CONTROL)----
-----

(defun take-picture-4 (Camera Window-stats List-of-objects view-stats)
 (cond ((or (null view-stats)
            (null list-of-objects))
        (return-from take-picture-4 'nil)))
 (setf *window-width* (third window-stats))
 (setf *window-height* (fourth window-stats))
 (setf *scale* (sixth window-stats))
 (setf *image-distance* (seventh window-stats))
 (send (eval Camera) :initialize-B Window-stats)
 (send (eval Camera) :move (- (first view-stats))   ; x
                           (- (second view-stats))  ; y
                              (third view-stats)    ; z
                              (fourth view-stats)    ; azimuth
```

```
                        (fifth view-stats)      ; roll
                        (sixth view-stats))     ; rotation
(loop for V in List-of-objects
      do (send (eval V) :initialize)
      do (send (eval V) :translate-and-euler-angle-transform (nth 6 view-stats)
                                                             (nth 7 view-stats)
                                                                    (nth 8 view-stats)
                                                                    (nth 9 view-stats)
                                                                    (nth 10 view-stats)
                                                                    (nth 11 view-stats))
      do (send (eval Camera) :take-picture V)
      do (let ((object-type (string-trim '"|0123456789 " V)))
           (cond ((string-equal object-type '"observer")
                  (let* ((obs-point (first (send (eval V) :get-transformed-node-list)))
                         (screen-point (send (eval Camera) :screen-transform obs-point)))
                    (send (eval (camera-*camerwindow* (eval Camera)))
                          :set-cursorpos (- (first screen-point) '30)
                                         (- (second screen-point) '5))
                    (send (eval (camera-*camerwindow* (eval Camera)))
                          :display-lozenged-string '"obs")))
                 ((string-equal object-type '"path")
                  (let* ((start-point (first (send (eval V) :get-transformed-node-list)))
                         (end-point
                           (first (last (send (eval V) :get-transformed-node-list))))
                         (screen-start-point
                           (send (eval Camera) :screen-transform start-point))
                         (screen-end-point
                           (send (eval Camera) :screen-transform end-point)))
                    (cond ((< '50000 (* *window-width* *window-height*))
                           (send (eval (camera-*camerwindow* (eval Camera)))
                            :set-cursorpos (- (first screen-start-point) '43)
                                           (- (second screen-start-point) '5))
                           (send (eval (camera-*camerwindow* (eval Camera)))
                                 :display-lozenged-string '"start")
                           (send (eval (camera-*camerwindow* (eval Camera)))
                                 :set-cursorpos (+ (first screen-end-point) '3)
                                                (- (second screen-end-point) '5))
                           (send (eval (camera-*camerwindow* (eval Camera)))
                                 :display-lozenged-string '"end")))))))))
```

```
FILE NAME: Kinematics.lisp
;
;  rotation and translation code  cs4452  17may88
;

(defun transpose (A)
 (cond ((null (cdr A)) (mapcar 'list (car A)))
       (t (mapcar 'cons (car A) (transpose (cdr A))))))
(defun dot-product (x y)        ;A vector is a list of numerical atoms.
 (apply '+ (mapcar '* x y)))    ;A matrix is a list of vectors representing
(defun cross-product (x y)
 (list (- (* (cadr x) (caddr y)) (* (caddr x) (cadr y)))
       (- (* (caddr x) (car y)) (* (car x) (caddr y)))
       (- (* (car x) (cadr y)) (* (cadr x) (car y)))))
(defun post-multiply (M x)      ;the rows of the matrix.
 (cond ((null (cdr M)) (list (dot-product (car M) x)))
       (t (cons (dot-product (car M) x) (post-multiply (cdr M) x)))))
(defun pre-multiply (x M)
 (post-multiply (transpose M) x))
(defun matrix-multiply (A B)
 (cond ((null (cdr A)) (list (pre-multiply (car A) B)))
       (t (cons (pre-multiply (car A) B) (matrix-multiply (cdr A) B)))))

(defun cycle-left (L) (mapcar 'row-cycle-left L))
(defun row-cycle-left (R) (append (cdr R) (list (car R))))
(defun cycle-up (M) (append (cdr M) (list (car M))))
(defun unit-vector (one-column length)
 (do ((n length (1- n))
    (R nil (cons (cond ((= one-column n) 1) (t 0)) R)))
   ((zerop n) R)))

(defun concat-matrix (A B)
 (cond ((null A) B)
       (t (cons (append (car A) (car B)) (concat-matrix (cdr A) (cdr B))))))
(defun augment (A) (concat-matrix A (unit-matrix (length A))))
(defun normalize-row (R) (scalar-multiply (/ 1.0 (car R)) R))
(defun scalar-multiply (a x)
 (cond ((null x) nil)
       (t (cons (* a (car x)) (scalar-multiply a (cdr x))))))
(defun solve-first-column (M)
 (do* ((L1 M (cdr L1))
     (L2 (normalize-row (car M)))
     (L3 (list L2) (cons (vector-add (car L1)
                                     (scalar-multiply (- (caar L1)) L2)) L3)))
    ((null (cdr L1)) (reverse L3))))
(defun vector-add (x y) (mapcar '+ x y))
(defun first-n (n R)
 (cond ((zerop n) nil)
       (t (cons (car R) (first-n (1- n) (cdr R))))))
(defun square-car (M)
 (do ((m (length M))
```

```
    (L1 M (cdr L1))
    (L2 nil (cons (first-n m (car L1)) L2)))
   ((null L1) (reverse L2))))
(setq A '((1 1 1) (2 1 2) (3 2 3)))
(setq B '((1 1 2) (1 2 3) (2 3 1)))
(defun ncdr (n L) (cond ((zerop n) L) (t (cdr (ncdr (1- n) L)))))
(defun ncar (n L) (cond ((zerop n) nil)
                        (t (cons (car L) (ncar (1- n) (cdr L))))))
(defun nmax-car-first (n L)
 (append (max-car-first (ncar n L)) (ncdr n L)))
(defun matrix-inverse (M)
 (do ((M1 (max-car-first (augment M))
       (cond ((null M1) nil)
             (t (nmax-car-first n (cycle-left (cycle-up M1))))))
   (n (1- (length M)) (1- n)))
  ((or (minusp n) (null M1)) (cond ((null M1) nil) (t (square-car M1))))
  (setq M1 (cond ((zerop (caar M1)) nil) (t (solve-first-column M1))))))
(defun max-car-first (L)
 (cond ((null (cdr L)) L)
       (t (if (> (abs (caar L)) (abs (caar (max-car-first (cdr L)))) L
              (append (max-car-first (cdr L)) (list (car L)))))))

(defun dh-matrix (cosrotate sinrotate costwist sintwist length translate)
 (list (list cosrotate (- (* costwist sinrotate))
             (* sintwist sinrotate) (* length cosrotate))
       (list sinrotate (* costwist cosrotate)
             (- (* sintwist cosrotate)) (* length sinrotate))
       (list 0. sintwist costwist translate) (list 0. 0. 0. 1.)))


(defun homogeneous-transform (azimuth elevation roll x y z)
 (rotation-and-translation (sin azimuth) (cos azimuth) (sin elevation)
 (cos elevation) (sin roll) (cos roll) x y z))


(defun rotation-and-translation (spsi cpsi sth cth sphi cphi x y z)
 (list (list (* cpsi cth) (- (* cpsi sth sphi) (* spsi cphi))
             (+ (* cpsi sth cphi) (* spsi sphi)) x)
       (list (* spsi cth) (+ (* cpsi cphi) (* spsi sth sphi))
             (- (* spsi sth cphi) (* cpsi sphi)) y)
       (list (- sth) (* cth sphi) (* cth cphi) z)
       (list 0. 0. 0. 1.)))

(defun A01 (d1)
 (dh-matrix 0 1 0 1 0 d1))
(defun A12 (d2)
 (dh-matrix 0 1 0 1 0 d2))
(defun A23 (d3)
 (dh-matrix 0 1 0 1 0 d3))
(defmacro A03 (d1 d2 d3)
 '(chain-multiply '((A01 ,d1) (A12 ,d2) (A23 ,d3))))
```

```
(defun A34 (theta4)
  (dh-matrix (cos theta4) (sin theta4) 0 1 0 0))
(defun A45 (theta5)
  (dh-matrix (cos theta5) (sin theta5) 0 1 0 0))
(defun A56 (theta6)
  (dh-matrix (cos theta6) (sin theta6) 0 1 0 0))
(defmacro A36 (theta4 theta5 theta6)
  '(chain-multiply '((A34 ,theta4) (A45 ,theta5) (A56 ,theta6))))
(defun A06 (d1 d2 d3 theta4 theta5 theta6)
  (matrix-multiply (A03 d1 d2 d3) (A36 theta4 theta5 theta6)))
(setq A6body '((0 0 1 0) (1 0 0 0) (0 1 0 0) (0 0 0 1)))
(defun homogeneous-transform1 (azimuth elevation roll x y z)
  (matrix-multiply (A06 z x y (+ azimuth pi) (- elevation (/ pi 2))
                                (+ roll pi)) A6body))
(setq B6body '((1 0 0 0) (0 0 -1 0) (0 1 0 0) (0 0 0 1)))
(defun homogeneous-transform2 (azimuth elevation roll x y z)
  (matrix-multiply (A06 z x y azimuth elevation roll ) B6body ))

; changes: D.H.Lewis 17 May 88

(defun unit-matrix (L)
  (loop for i from L downto 1
        collect (loop for j from L downto 1
                      when (equal i j)
                        collect 1
                      else collect 0
                    finally)
    finally))

(defun chain-multiply (L)
  (cond ((equal (length L) 2) (matrix-multiply (eval (first L)) (eval (second L))))
                              (t (setq temp (matrix-multiply (eval (first L)) (eval (second
L))))                                 (chain-multiply (push 'temp (cddr L))))))
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center — 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Library, Code 0142 — 2
   Naval Postgraduate School
   Monterey, California 93943-5002

3. Cruise Missiles Project (PMA 281) — 2
   Naval Air Systems Command Headquarters
   Washington, DC 20361-1014

4. Neil C. Rowe, Code 52 — 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5100

5. LT David H. Lewis — 2
   Commanding Officer
   Naval Surface Force Pacific
   Readiness Support Group
   San Diego, Box 124
   Naval Station
   San Diego, California 92136-5126